AD-A060 210 ARMY MILITARY PERSONNEL CENTER ALEXANDRIA VA F/G 9/2
ARIAN -- AN IMPLEMENTATION OF A MICROCOMPUTER OPERATING SYSTEM.(U)
MAY 78 R L CONN

UNCLASSIFIED NL

1 OF 2
AD A060 210

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

ARIAN -- AN IMPLEMENTATION OF A MICROCOMPUTER OPERATING SYSTEM

5. TYPE OF REPORT & PERIOD COVERED

Final Report, *Thesis*
May 1978

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

1LT Richard L. Conn
311-48-5360

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Student, HQDA, MILPERCEN (DAPC-OPP-E)
200 Stovall St, Alexandria, VA 22332

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

HQDA, MILPERCEN, ATTN: DAPC-OPP-E
200 Stovall St, Alexandria, VA 22332

12. REPORT DATE

May 1978

13. NUMBER OF PAGES

184+3 (187)

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Document is a thesis, filed with the Graduate College of the University of Illinois at Urbana, IL

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

operating system; microcomputer; microprocessors; assembler; file system; floppy-disk; computer system

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ARIAN is a microcomputer operating system developed for use with the Z80 microprocessor. A floppy-disk based operating system, ARIAN was designed to be used as a software development system. Its features include a dual file system for memory and disk files, a built-in assembler input line and intra-line editing facilities, and a host of utilities which may be used for system control and programm debugging.

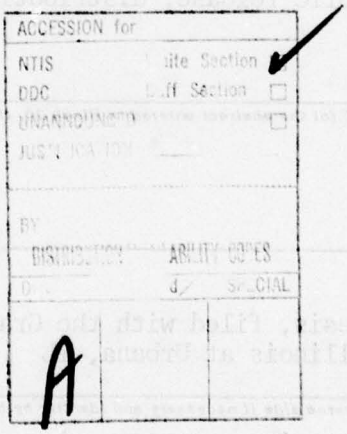
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

391 191

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Continuation of Block 20:

Although ARIAN was designed specifically for the Z80 microprocessor, the basic concepts learned during its creation and implementation are of general application. These concepts are discussed in the first part of the thesis, while the following sections describe ARIAN in detail and function as a users' manual.



# ARIAN -- AN IMPLEMENTATION OF A MICROCOMPUTER OPERATING SYSTEM

1LT RICHARD L. CONN
HQDA, MILPERCEN (DAPC-OPP-E)
200 Stovall Street
Alexandria, VA 22332

Final Report, May 1978

Approved for Public Release; Distribution Unlimited

A thesis submitted to the Graduate College, University of Illinois at Urbana-Champaign, Urbana, Illinois, in partial fulfillment of the requirements of the degree of Master of Science in Computer Science.

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

May, 1978

WE HEREBY RECOMMEND THAT THE THESIS BY

RICHARD LEE CONN

ENTITLED ARIAN -- AN IMPLEMENTATION OF A MICROCOMPUTER OPERATING SYSTEM

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

R H Campbell

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairman

† Required for doctor's degree but not for master's.

# ARIAN -- AN IMPLEMENTATION OF A MICROCOMPUTER OPERATING SYSTEM

BY

RICHARD LEE CONN

B.S., Rose-Hulman Institute of Technology, 1976

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1978

Urbana, Illinois

## Acknowledgments

I would like to thank Professor Roy Campbell for the effort he devoted to this project and the faculty and staff of the Department of Computer Science of the University of Illinois for their general support. I would also like to thank Professor Alfred Weaver of the Department of Computer Science of the University of Virginia for effort expended during the early days of Project ARIES and his interest and suggestions made during the creation of ARIAN.

# ARIAN -- AN IMPLEMENTATION OF A MICROCOMPUTER OPERATING SYSTEM

## TABLE of CONTENTS

# CHAPTER 1

## INTRODUCTION

Operating systems are instrumental parts of every commercial computer system. They control the resources of the computer system and monitor the execution of programs within the computer system. They handle input/output (I/O) between computer and its users and within the computer itself. They support the high-level languages which often are employed by users of the computer to help solve their problems. Finally, operating systems occasionally permit one computer to converse with another.

In 1971-1972 the first complete microprocessor, the INTEL 8008, was marketed. This microprocessor, as well as the ones to follow shortly, were initially employed only by industry to computerize the many automation operations which were previously done by discrete components or integrated circuits and wiring. The microprocessor greatly reduced

the size and price of the machines which performed the automation operations, and it added a flexibility which was not previously known; it gave the manufacturer the ability to program his automations. With this programming capability, the value of the automation was greatly increased and the cost was greatly reduced. Now, a machine could be reprogrammed if the specifications of the item it was producing changed; another machine did not have to be built. A machine could even be reprogrammed to build an entirely different item. The possibilities were numerous; the limitations placed on the application of microprocessors in industry were due largely to the creativity of the scientists and managers. In 1977, with several hundred thousand microprocessors being produced annually, business and industry utilized by far the largest percentage of these microprocessors -- 90 percent [15].

But very little was done in the creation of operating systems for the microprocessor. The microprocessor was a general-purpose device which could be programmed for a specific application, and all the operating system which it required was contained within its application program. Software for the microprocessor consisted largely of cross-software which ran on the larger computers and produced machine code output in some form which could be transported to the microprocessor and loaded into its memory. This cross-software, however, was quite expensive to use, and software development for the microprocessor was slowed by the transportation and loading problem. As a result, the emphasis slowly began to shift from cross-software to

resident-software. INTEL Corporation was a front-runner in the field, and it was also one of the first to put a serious effort into developing resident software for the microcomputer.

The author came to the University of Illinois in 1976, and at this time, the Department of Computer Science owned two microcomputer systems (the MUMS Project [2]) and a cross-assembler which resided on the university's DECSYSTEM-10. Each microcomputer system possessed a PROM-resident monitor which supplied the user with the basic abilities of examining and altering memory, running programs stored in memory, and loading paper tape. All major development software, i.e., the cross-assembler, was resident on the DECSYSTEM-10.

The problem addressed by the author in this thesis is the creation of resident software for the 8080, and later Z80, microprocessor. Such software was designed to be used for software development on a microprocessor for a microprocessor. An operating system had to be created which would allow the user to create assembler-language programs, edit the programs, assemble them, and execute and debug them. It was also hoped that a higher-level language, such as FORTRAN or BASIC, could be implemented on the microcomputer so that the microcomputer could also become a computational tool.

## The Problem in General

The problem of creating an operating system with file editing and assembly capabilities on a microcomputer is not trivial. No models of such a system were available for the author to study at the time of formulation of the problem, and the hardware facilities of the MUMS project proved inadequate to address such a problem. As a result, the author opted to build his personal microcomputer system which would have the required facilities to address this problem. The history of this project, which was later called Project ARIES, is given in Chapter 2, and it will not be discussed here.

The problem of creating an operating system was divided into two parts: (1) building up the hardware facilities and learning about the operation and programming of microprocessors and microcomputers, and (2) designing and programming the operating system itself. The first part, that of building the microcomputer, was done solely by the author at his personal expense. As time progressed and more was learned about microprocessors, the nature of the equipment required for the problem was determined, the required equipment was purchased, and the new equipment was integrated with the microcomputer.

The second part of the problem, that of designing and programming the operating system, was solved through the acquisition of skills and

knowledge gained by designing several small monitors, applications programs, and operating systems. As more work was done in programming such systems, a "feel" and expertise was developed by the author toward his microcomputer; such expertise greatly enhanced the creation of the operating system. The limitations of the microcomputer became very well known, and a programming methodology was developed. All of this ultimately led to the creation of a microcomputer system and an operating system which were uniquely integrated; one was designed and configured around the other. Such an integration is not always exhibited in commercial computer systems. Commands were implemented to conform with hardware facilities, and hardware facilities were designed to permit commands to function.

This integration is probably the greatest contribution of ARIAN (the operating system) and ARIES-I (the microcomputer). Throughout the work on Project ARIES, the author has gone from microcomputer systems applied to specific applications to general-purpose microcomputer systems to ARIES-I, an integrated special-purpose microcomputer system. Perhaps the best summary of Project ARIES is the statement that microcomputers are best applied to specific applications; the larger computers are better suited to the general-purpose applications.

## The Details of the Operating System

The problem of creating ARIAN was divided into six partitions: (1) input line editing, (2) command structure and parsing, (3) command execution, (4) file manipulation and mass storage, (5) assembly of files, and (6) interfacing to external programs. The following addresses the questions associated with these partitions.

After creating other monitor and operating systems (see Chapter 2), it was decided that all input to ARIAN should pass through a line editor. But what should the editing commands be like, how should they be implemented, how should the input line buffer be constructed, and what character set should be used? These questions are discussed in Chapter 3.

What commands should be employed by the operating system, what should the command structure be, and how should parsing be done? These questions are also discussed in Chapter 3.

How should the commands be executed and should the user be given control over the commands? Again, Chapter 3.

What should be the structure of files for ARIAN, should memory residency of files be required, how should disk files be controlled, how

can disks and tapes be controlled, and how should the file editor be implemented? These questions are discussed in Chapter 4.

What should the assembler be like and how should it be implemented? The assembler is discussed in Chapter 5.

Finally, how should an interface be established with other programs, how should parameter passing be done, what form should operating system support take, and how should the operating system be structured to run with other programs? These questions are discussed in Chapter 6.

Hence, there are many questions to be discussed throughout this thesis. Many of their answers, however, are a matter of personal preference. Many of their answers must be considered in terms of the microcomputer environment, an environment in which the set of laws which are assumed in the environments of the larger computers does not exist. And, many answers must be based on hardware availability. There is no clear solution to any question presented above.

## Summary

This thesis, then, discusses many of the questions presented above. It describes ARIAN. But ARIAN is not an ultimate answer. ARIAN is a stepping stone; it is the first attempt on the part of the author to create a large-scale microcomputer operating system.

ARIAN is one of the first generation of microcomputer operating systems. Its success suggests that large scale operating systems can be constructed for future microprocessor systems. It is believed that the problems examined or solved in this thesis will contribute to a better understanding of the requirements of microcomputer operating systems and to the future development of more sophisticated systems. IDIC.

# CHAPTER 2

## PROJECT HISTORY

The project which produced ARIAN, Project ARIES, lasted for about one and a half years from December of 1976 to April of 1978. Project ARIES was implemented and maintained by the author with a number of goals in mind, some of which include: (1) to learn about how to use and design hardware and software for microprocessors, particularly the INTEL 8080 and ZILOG Z80, (2) to acquire a personal microcomputer system, (3) to develop and work with cross software for microcomputers, and (4) to develop a floppy-disk-based operating system for the Z80 microcomputer.

The author was introduced to the MUMS project [2] at the University of Illinois shortly after his arrival in August of 1976. He worked with this project until December of 1976, when he purchased the initial hardware configuration of ARIES-I, the Z80-based microcomputer system on which ARIAN currently resides.

The initial hardware configuration was minimal for software development with a microcomputer. It consisted of an IMSAI 8080 mainframe with front panel, power supply, 8080 MPU and supporting circuitry, 4K of RAM, one serial I/O port and three parallel I/O ports (one of which was on the front panel of the IMSAI). With this hardware configuration and an ASR-33 teletype, the author was able to develop the Bootstrap Monitor System (BMS) [5].

BMS is a very basic monitor, but the author considered it to be superior to the current MUMS monitor at that time. BMS provided memory examine, deposit, and breakpoint routines similar to those supported by the MUMS monitor. In addition, BMS provided an input line editor, a paper-tape load program callable by a BMS command, and a small set of system utility subroutines which could be called by the user to support his programs. At this time the author purchased a BYTESAVER (a 2708 PROM programmer with sockets to hold up to eight 2708 PROMs in memory space) and placed BMS (a 1K program) into a PROM. Bootstrapping the microcomputer was now a simple matter of examining the starting address of BMS and toggling the RUN switch on the front panel.

After BMS was completed, the author then undertook a much larger task -- the task of writing a larger operating system. This OS was named the Advanced Operating System (AOS) [6]. This 8K program, developed using cross software on the university's DECSYSTEM-10 and paper tape to load the program into the author's microcomputer, proved to be a very good learning tool for the systems to come.

Many new concepts were employed in the design of AOS; some of these features included: (1) an input line editor which permitted editing of the command line in a manner similar to the SOS editor on the DECSYSTEM-10, (2) three-character command names with a variable-length operand field, (3) a disassembler, (4) block memory examine and ASCII print routines, and (5) the ability to redefine the I/O parameters of the operating system. These new concepts posed many significant problems in software design for the author, and the experience of writing AOS later proved invaluable in the creation of ARIAN. Documentation on AOS currently exists only in its source listing and in lesson 'os8080' on the PLATO computer system of the University of Illinois.

The next major software advancement was the development of the DECLINK program by the MUMS project. This interrupt-driven program permitted a MUMS user to log into the university's DECSYSTEM-10, timeshare through MUMS microcomputer, assemble programs on the 10 and download them to the microcomputer, and execute the downloaded program. This was a dramatic step forward which reduced the software development time by about a factor of three. A user was no longer required to assemble a program on the 10, have the 10 punch a paper tape, carry the paper tape to the MUMS unit, load the paper tape (perhaps after several tries) into the microcomputer, and then execute the program. The author soon followed the lead of the MUMS project and wrote a polling link program, put it into a PROM (thereby avoiding the need to load a paper tape whenever the link was to be used), purchased a dual serial I/O

board to control a modem, and used a modem at the university to timeshare with the DECSYSTEM-10.

The next monitor system written by the author was created as an advanced version of BMS. This system, called the Monitor Command System (MCS) [1], was written employing many of the lessons learned while writing AOS and the polling link program. MCS was designed to be a 1K program which supported timesharing and all the BMS functions except paper tape loading. It contained an improved input line editor which the author considered to be superior to the old BMS line editor, more system subroutines which were accessable by the user for his programs, and many programming optimizations which were learned by the author while writing AOS. MCS allowed the author to do all his future work in a timesharing mode without using paper tape; this was a significant step forward.

This version of MCS was to later become known as MCS V1.2, and MCS continued to mutate until it reached its present state, MCS V1.6. Some of the significant changes undergone by MCS during its evolution included (1) the incorporation of an alternate command set into MCS, (2) the creation of DEBUG, an extensive diagnostic package to be used as an alternate command set of MCS, and (3) finally the creation of UTILITY and the extensive dependence of MCS on UTILITY for support. DEBUG and alternate command sets are described in the MCS USERS' MANUAL [1], which reflects MCS V1.4. MCS V1.6 and UTILITY are described in the ARIAN SOFTWARE LIBRARY, VOLUME 1 [12].

In August of 1977, the DECSYSTEM-10 was replaced by a CYBER-175 computing system. At this time, the author successfully transferred the "universal" cross assembler from the old DECSYSTEM-10 to the DECSYSTEM-10 at the Coordinated Science Laboratory and made it operational. Also, the author was given access to the MAC80, INTERP80, PLM, and ASSEMB systems which were being placed on the CYBER and retained copies of these systems on tape. Later, when these systems were purged from the CYBER, the author was able to reconstruct and maintain them from his tape and continue his work.

It was under these reconstructed systems that ARIAN was created. The expense became prohibitive as ARIAN grew in size, and, when George Lehmann, a fellow graduate student, wrote a Z80 cross assembler [14] for the CYBER, the author converted ARIAN to Z80 mnemonics and gave a copy to Mr. Lehmann. Work progressed on ARIAN, and work was also done on Lehmann's version of ARIAN concurrently. Lehmann's personal system is a cassette-tape oriented system, and two different versions of ARIAN rapidly began to develop; one version was an extension of the original disk version, and the second version is Lehmann's tape version. The two versions are now distinctly different in many ways, but they retain many common data structures and commands. The author's floppy-disk-based version of ARIAN was finally completed in April of 1978.

# CHAPTER 3

## THE ARIAN COMMAND SYSTEM

The command system of ARIAN parses and executes all commands given to ARIAN by the user. Based on a combination of the characteristics of the MCS ([1]), AOS ([6]), SCS ([10] and [11]), the Northstar monitor ([7]), and CUTER ([8] and [9]) command systems the ARIAN command system was designed with two goals in mind: (1) to human-engineer the commands in order to make them easy for the user to type and remember and (2) to minimize the parsing and execution overhead.

With these goals in mind, the command system was implemented as a series of subroutines. This "structured" approach was taken to permit later modification of ARIAN and substitution of whole functions within ARIAN. Each phase of command evaluation (input line editing, command parsing, and command table search) as well as each command is implemented as a subroutine. Parameters are passed between each

subroutine through the system scratchpad RAM to eliminate the need to reserve MPU registers which may be required for other uses, and the entire input line in its edited form is also preserved. This chapter of the thesis describes the ARIAN command system in detail.

## The ARIAN Input Line Editor

All commands are input to ARIAN through its input line editor. This subroutine permits the user to enter his commands as strings of ASCII characters typed on the I/O device currently driven by the redirectable I/O driver (discussed later). These strings are stored in the input line buffer (IBUF) and passed to the calling program in this buffer.

The input line editor in ARIAN is almost an exact duplicate of the editor in MCS [1], which was in turn derived from BMS [5]. It was designed with three constraints in mind: (1) to be human-engineered and use editing control characters which are mnemonic in nature, (2) to be implemented so that it could be used on both hard-copy and soft-copy terminals, and (3) to minimize the size of the editing program. Speed

is not of concern in the case of an input line editor since it is interfacing with a human. The editing functions implemented by the editor are deletion of the previous character (using either a ctrl-H or the rubout key), deletion of an entire line (using the escape key), continuing the line on the next physical line of the I/O device (using the line feed key), and terminating entry of the line (using the carriage return key). More advanced editing features such as the editors of AOS [6] and the Northstar monitor [7] were not implemented due to the constraint of minimizing the size of the editing program.

Ctrl-H and rubout were both implemented to allow for both hard-copy and soft-copy terminals. Ctrl-H echoes as a physical backspace of the cursor on the user's terminal, while rubout echoes as the deleted characters enclosed in backslashes (see the users' manual in Appendix I for an explanation of the editing commands).

No control characters (except ctrl-H) were used because of the physics of the keyboard. It is felt by the author that the need to hold down two keys to implement an editing command should be avoided if possible; the editor should make the communication of its commands as simple as possible, perferably using only one key. Ctrl-H was chosen for backspace because some terminals also have a backspace key which transmits this character. Hence, all editing commands may be transmitted in one keystroke.

## The ARIAN Command Parser

Like the input line editor, the command syntax and the command parser were designed around the constraints of human-engineering and size minimization. In order to determine exactly what the command syntax would be, an analysis was made of the commands to be implemented. As a result, it was decided by the author that the arguments to be passed to the execution routines must consist of at most one file name and three numeric values. The exceptions which require more than one file name, like the rename command, must supply their own input routines for the second filename.

Since the input numbers may be either decimal or hexadecimal (like, decimal for line numbers and hexadecimal for memory locations), the input numbers are passed as hexadecimal values (interpreting their strings to represent hexadecimal numbers) and ASCII character strings for use with the file editor (string comparison for line insertion). Therefore, three buffers are used by the command parser; these buffers are ABUF (ASCII buffer for numeric values), BBUF (binary buffer for numeric values), and FBUF (file buffer for file names).

The next decision to be made was what format the commands themselves should take. This decision was made based on the following

constraints: (1) the commands should be mnemonic in nature, (2) they should be reasonably short to simplify look-up and parsing, (3) they should allow additional options to be specified (like, formatted listing, unnumbered listing with the same LIST command), and (4) they should allow the user to append characters to the command name to improved its readability (like, EXAM may be entered as EXAMINE). As a result, a command structure similar to that employed by SCS (see [10] and [11]) was chosen.

The command structure consists of a four-character keyword followed by an optional special character (to specify command variants) and an optional group of miscellaneous characters to improve readability. The choice of four significant characters was arbitrary; three characters are required to distinguish between commands like EXIT, EXAM, and EXEC, so it was decided to use four characters to meet most conceivable extrapolations (like, EXEC and EXEG).

Finally, the general format of the command was determined. It was decided by the author to separate each command and argument by a string of one or more blanks, the file name or string name must be the first argument, and the numeric arguments must follow. The SCS convention of enclosing the file name in slashes is considered to be poor human engineering, so the author opted to not enclose the file name in slashes and require that the numeric arguments, particularly the first numeric argument, begin with a digit. Hence, the command structure described in Chapter 1 of Appendix I is the command structure employed by ARIAN.

This command structure reduces the size of the parser, permits command variants, permits the required arguments, and is human-engineered.

## The ARIAN Command Execution Section

The command execution section of ARIAN is designed with future modifications, flexibility, and minimum memory utilization in mind. This section of ARIAN is invoked after the command is acquired and parsed; the command name resides in the first four bytes of the input line buffer (IBUF).

All ARIAN system commands are stored in a table which consists of the command name and the address of the subroutine within ARIAN which executes the command function. Command execution involves looking up the command in the table and calling the routine located at the address specified by the table entry.

A design extension incorporated into the command execution section is the ability to execute user-defined commands. These commands,

entered into the system by the user through the CUST command, permit the user to create his own set of commands and overwrite the system commands as desired. Customized commands are implemented by simply keeping another table of command names and execution addresses. This customized command table is scanned first by the command execution section, and, if a match is found, the specified routine is immediately executed; if no match is found, the command execution section then passes on to the system command table.

As can be seen by the above, system commands can be overwritten by simply creating a customized command of the same name. A match will be found in the customized command table, and this new command will be executed without a search being done on the system command table and the corresponding system command being executed.

Hence, the ARIAN command execution section lends itself easily to modification and extension. System commands can be redefined and their names can be changed by simply altering the system command table, thereby making an alteration in ARIAN itself. Also, a user can create additional commands by simply using the CUST command and its variants. Complete control of the customized command table is given to the user through the CUST command.

# CHAPTER 4

## ARIAN FILES and FILE STRUCTURES

An instrumental part of ARIAN, as well as other operating systems, is the ability to manipulate files. ARIAN has several file-manipulation capabilities, including the abilities to create, modify, delete, verify the structure of, and catalog files in memory and on disk.

A file, as defined for this paper, is a logical unit of textual or binary information. It consists of two parts: its contents (the information contained within it) and its structure (how the file appears in memory). Its contents are completely controlled by the user; he may place any type of information, like text or binary code, into it. Its structure, however, is controlled by ARIAN. Although the user controls the information contained in the file, ARIAN controls the form that information takes as it appears in memory.

The way an operating system structures its files affects to a large degree the operation of that system, especially in the microcomputer environment. The microcomputer environment is greatly limited in resources as compared to the environment of more conventional computer systems. Memory in a microcomputer is at a premium, normally expandable to a maximum of 64K; execution speed in a microcomputer is also at a premium, with instruction executions ranging from two to ten microseconds and data transportation from such mass storage devices as floppy disk running at 250,000 baud and cassette tape running at 300 to 1200 baud. It is obvious, then, that files must be structured to consume as little memory and CPU overhead as possible. This chapter discusses the file structures considered and the file structure finally chosen to be used in ARIAN.

## Proposed Text File Structures

The first question to be considered in proposing the structure of a file concerns in what medium or mediums the file will reside. If the file resides totally in memory, a sequential or a linked structure would be appropriate; if the file resides totally on disk, a sequential or a linked-block structure should be considered. Since ARIAN deals with a microcomputer environment, it was decided to have the file currently being edited or created reside in memory.

Memory residence of the working file was chosen for a number of reasons, almost all of which are directly related to the microcomputer environment:

1. User convenience is the first and most significant reason for this choice. A design decision was made that the best type of human-engineered file system should inconvenience the user and cause him to wait on a system peripheral as little as possible. Such would be the case if the file system relied in any extent on the disk for support while editing or creating a file. Disk startup, head movement, data transfer, and verification for relatively large blocks (1K to 4K) of sequential data can take as much as one to two seconds, and this time lag is considered to be unacceptable. Also, since the floppy remains dormant when not in use, disk and head wear are reduced by minimizing the disk accesses.

2. ARIAN was initially conceived to be a software development system for microcomputer programs. As discussed in the CS397 notes [15], approximately 90% of the microprocessors currently in use operate in an industrial environment on programs whose

length is 1K or less; hence, the size of the program is almost inconsequential, and, through experience, the text for such a program would be less than 10K in length. ARIAN, operating on ARIES-I under a 32K memory environment, easilty supports a program text of 20K in length. This is more than adequate to meet the needs of the industrial applications and many conceivable user applications. Only large-scale software development cannot be supported in full under ARIAN; it can only be supported if the software is partitioned into smaller, self-contained structures. This need for partitioning is not considered to be a detrement to the system.

3. The speed of the microprocessor itself, while much slower than a conventional CPU, is relatively fast when used in an environment of human interaction. Block movements of data, for instance, are very fast (can be done in one instruction on ARIES-I's Z-80), and text editing functions such as insertion and deletion can be done with no user inconvenience when the file resides in memory; a file residing partly on disk, however, may require as much as a two second delay between each insertion. Under this condition, memory residency is definitely preferred.

4. Finally, as in all communication media for microcomputers, the disk is susceptable to more errors in data transfer than memory. Hence, memory residency reduces the error checking and recovery overhead which must be used in a disk system.

Now that it has been decided to make the working file reside totally in memory, the second problem to be addressed concerns what the physical structure of the file should be. Three constraints were met in addressing this question: (1) the file should consume as little memory space as possible with extraneous pointers and non-data information; (2) the file should be structured in such as way as to permit rapid manipulation by the microprocessor and minimize processing wait time at the human's end; and (3) the file should contain some error-checking properties that permit validation of the data in the file. With these

constraints in mind, linked and sequential structures were considered to be the most feasable structures to employ.

A linked structure, however, provides several drawbacks. First, there is the overhead of the pointers, with two bytes required for each pointer. The linked structure must have forward-linking and optionally backward-linking capabilities, and these links should extend to allow links within a line which would result from editing a given line. Secondly, in order to support fragmented lines, an additional byte may be required on each pointer to indicate end of line, end of file, or end of line segment. Under this scheme, three bytes are required for each line fragment, making the amount of non-data in a file grow in multiples of three as more editing is done. Thirdly, a relatively complex memory management scheme would be required to reduce loss of available memory space due to this line fragmentation. Fourth, a minimum of one additional byte would be required on each line if any type of file validity checking is to be done. Finally, a workspace would be required to sequence the file for storage on disk. The disk system employed on ARIES-I permits data to be transferred to and from the disk in 256-byte blocks only, and any linked files must be sequenced in this block format for storage.

Sequential storage of file data, on the other hand, shows several advantages and few drawbacks. First, non-data overhead is minimized. In the file structure used in ARIAN, one byte is used at the beginning of each line to give a count of the number of bytes consumed by the line

(the number of bytes of data plus the end-of-line mark and the character count itself). This count was placed in the line for validity check only, and it is sufficient to insure security of the file. Hence, only two bytes of non-data are required per line in the file, while a minimum of three bytes are required using a linked structure. Secondly, processing overhead is required for insertions, deletions, and editing of lines within a file. Although this seems like a drawback, the microprocessor in ARIES-I is able to handle the massive movements required for such operations without any noticable delay to the user. Finally, the memory management and disk transfer problems encountered with a linked structure are not encountered with this sequential structure. No memory or line fragmentation occurs since the file exists as sequential strings of bytes at all times. Also, the file may be stored on disk exactly as it resides in memory without the buffering and restructuring required of the linked structure.

The only real advantage of the linked structure over the sequential structure is that the linked structure permits handling files larger than memory by loading portions of the file at a time and then linking it together in blocks on disk. Since total memory residency of the working file was decided upon by the reasons given above, however, this advantage is negated.

## Binary File Structure

Binary files are much more fixed and inflexible than text files; they must reside at a specific location in order to execute, their structure is precisely defined and must not be altered, and editing without knowing exactly what the bytes of the file represent is impossible. For these reasons, ARIAN does not handle binary files directly. A binary file is defined only by its entry in the local binary file directory (discussed below).

## Local Files, Disk Files, and Directories

ARIAN divides files into three general catagories: (1) local text files, (2) local binary files, and (3) disk files. A local file is a file which resides in memory, and a disk file resides on disk.

Following the philosophies of the CDC CYBER computer systems and the ALS-8 microcomputer operating system, ARIAN has a local text file facility incorporated into it. It maintains a directory of up to ten

text files residing in memory simultaneously, and it permits editing of one of these text files, designated as the primary file. The primary file is the only file that can be modified or operated on; all other text files merely reside in memory for future designation as the new primary file.

A local text file consists of a line number from 0 to 9999 followed by a blank and a string of text. The decision to number all lines of a text file was made solely on the personal judgment of the author. Numbering of lines provides an easy way to control the order of the lines in the file, copying lines and duplicating lines within a file (by changing line numbers only in the editor), and inserting and deleting lines within a file Other editors, such as BOSS of the University of Illinois and SOS and TECO of Digital Equipment Corporation, were examined, and it is felt that the numbered format has definite advantages over the unnumbered format. The only major advantage of the unnumbered format was that it did not require the user to enter a line number each time he entered a new line, but this advantage was circumvented by the incorporation of the APND and INS commands into the editing section of ARIAN.

The directory for the local text files provides a minimum of information to ARIAN. Each entry contains the name of the file, a two-byte pointer to the beginning of the file, a two-byte pointer to the end of the file, and a four-byte ASCII representation of the maximum line number in the file (kept for editing purposes). With the limits of

the file defined precisely in the directory, ARIAN has all the information it needs to modify the file and save it on disk.

The directory for the local binary files, on the other hand, gives only the name of each file and pointers to its beginning and end. Again, this is the minimum amount of information necessary to maintain a binary file.

Disk files, however, are an entirely different case. A disk file consists of one or more 256-byte blocks stored sequentially on disk. This structure is optimal for data transfer between disk and memory. Each entry in the disk file directory, which is stored in the first four blocks of the disk, contains the name of the file, the disk address of the first block of the file, the length of the file in blocks, the type of the file (text, binary, BASIC program, or BASIC data), and other information such as the execution address if it is a binary file and the load location if it is a BASIC program file.

The directory structures, therefore, are minimal. Protection mechanisms were not considered for ARIAN since it runs in a microcomputer environment which is designed for a single user. Protection of files can be accomplished by the user simply removing the disk he has been using and taking it with him.

## Memory and Disk File Management

The locating, verificiation, and monitoring of local text files is done through a memory manager in ARIAN. This subroutine keeps a dynamic map of workspace memory (where the workspace boundaries are defined by ARIAN and may be redefined by the user) and monitors the growth of text files within the workspace. Whenever a file is designated as primary, the memory manager moves this file to the physical end of the last non-primary local file, compresses the local files until no gaps remain between them, and permits the user to modify the new primary file. The memory manager also constantly monitors the growth of the primary file and warns the user if it exceeds the upper workspace boundary.

Unlike ALS-8 and SCS, ARIAN never requires the user to specifically control the location or position of his text files. The memory manager does that for him. Also, in assembly and automatic file execution, the memory manager controls by default the location of the binary file generated, so the user need never be concerned with where his binary files are being created.

Disk files are also managed to some extent by ARIAN, but, unlike the memory management subroutine, disk management does no compaction. Whenever a file is created on disk, the disk manager first checks to see if a file already exists under the specified name. If it does not, the

disk file directory is searched for the largest disk location at which a file is located and the new file is placed immediately after this file. If the specified file already exists on disk and the user wishes to replace it with this new file, the size of the old file as it exists on disk is checked against the size of the file that is to replace it. If the size of the new file is less than or equal to the size of the old file, the old file is written over and its directory entry is modified to reflect the size of the new file. If the size of the new file is greater than that of the old file, the old file entry is deleted and the new file is stored as if there was no old file (at the end of the disk file area).

No compaction of disk files is done by ARIAN implicitly. Compaction is a time-consuming task, and a human-engineering decision was made to allow the user to explicitly compact the disk files when he felt it was necessary. In this way the user does not have to wait on ARIES-I to compact his disk files unless he has specifically instructed ARIAN to do so. Also, compaction is not necessarily done every time a hole develops in the disk file space, and disk and head wear are reduced.

# CHAPTER 5

## THE ARIAN ASSEMBLER

The assembler is a major part of ARIAN, and it satisfies one of the main purposes of ARIAN -- to provide local assembly capability. It was designed with three constraints in mind: (1) it must be human-engineered, (2) it must be INTEL compatable, and (3) it must provide listing and diagnostic facilities.

Ease of use was of primary consideration in the human engineering of the assembler. To facilitate this, the assembler was designed to accept free-formatted source files. The only constraints placed on the user were to always start labels in column one and to never start op codes in this column. Each field of an assembly line must be separated by at least one blank, and the user may start a comment with any character (if it does not start in column 1).

Label length was a point of concern in the creation of the assembler. The labels had to be long enough to be mnemonic but still be short enough to be handled with a reasonable amount of memory overhead. The choice was made through trial; the author wrote several programs using different label lengths and chose six-character labels to be optimal. This label length, however, only refers to the significant characters; a label may be of arbitrary length provided it does not fill the input line buffer.

The assembler is based on the assembler in IMSAI's SCS [10,11]. Many design principals, such as the source line parsing and symbol table structure, were duplicated in the ARIAN assembler. The ARIAN assembler, however, is not a simple INTEL assembler, as the SCS assembler is. The ARIAN assembler supports several Z80 extensions. It was decided not to use ZILOG mnemonics for the Z80 extensions, but to use mnemonics resembling extensions to the INTEL standard. This choice was made because the parsing of the ZILOG mnemonics is an exceptionally complicated process, and simple INTEL extensions could be implemented on the assembler without extensive modifications.

The Z80 instructions chosen to be implemented on the ARIAN assembler were chosen for their usefulness and ease of implementation. All the relative branch and repetitive instructions were chosen because of their usefulness and tendency to conserve memory (being one- or two-byte instructions). Several special Z80 instructions, such as "NEG" and "OUT (C),A", were also chosen for their usefulness. The bit test

and set instructions were not chosen because they are memory-space consuming, difficult to implement in the assembler, and not considered to be exceptionally useful, especially since their functions can be implemented using other instructions.

A unique feature of the ARIAN assembler is a set of system-reserved symbols which allow the user to call upon system subroutines by name. This feature is extremely useful since it eliminates the need for the user to write some of the common utility routines, such as I/O and conversion routines, every time he wishes to use them. The user may, provided he knows the register conventions for the routines he uses, simply call the desired routines by name.

Two problems faced by the author in providing these symbols were how to name the symbols and what routines to give the user access to. The first problem was solved by simply starting the names of all the reserved symbols with a common letter; a "Z" was chosen. The rest of the symbol would be mnemonic in nature. With this convention, the user could resolve most symbol conflicts (concerning the use of reserved symbols) simply. The second problem, that of deciding which routines to give the user access to, was resolved by the author chosing these routines based on experience. The general routines used most frequently in the operating system and by the author were chosen.

Another unique feature incorporated into the assembler is the relative branch and call feature. Through the use of the ZJRL and ZCRL

symbols (jump relative long and call relative long), as well as the branch relative op codes, the user can write totally relocatable programs. The relative long symbols permit next-instruction-relative addressing over 64K of memory. The normal Z80 branch relative op codes extend over just 256 bytes of memory. This relocation feature is considered to be of significance by the author.

The symbol table of the ARIAN assembler is its final unique feature. This symbol table allows up to 384 symbols (occupies 3K), and its size was chosen to allow the user to work with very large programs. Most commercial resident assemblers allow only 256 symbols, and this size is considered to be prohibitive. Very large programs, such as small operating systems, cannot be assembled by these assemblers. The extended size of the ARIAN symbol table, which the user can reset to any size up to 65,536 symbols (see the ARIAN source listing), is a very nice feature, permitting work on arbitrarily large programs. The size of memory becomes the only restriction.

In summary, the ARIAN assembler is a significant contribution to microcomputer-resident assemblers. It permits a very large number of symbols, it allows easy access to many common utility subroutines, it gives the user the ability to write totally relocatable programs, and it is human-engineered for ease of use. Its only real restriction is that it is memory-resident and requires the files it assembles to be memory-resident. On ARIES-I, however, this is not considered to be a restriction, since ARIAN has almost 25K of text workspace. Also, as

noted in the CS397 notes [15], most programs (90%) are 1K or less in size.

# CHAPTER 6

## ARIES-I and ARIAN SYSTEM INTERFACING

As in many operating systems, the interfacing of user programs and system programs occasionally becomes a problem. In ARIES-I and ARIAN, however, this problem is not significant.

User programs and system programs are executed by ARIAN and MCS [2] in exactly the same way -- as subroutines. Knowing this, the user can effect a clean return to the operating system by simply balancing the stack and executing a return instruction when his program is finished. Three entry points, one of which is given as the reserved label ZEOR, may also be used to effect a clean return to ARIAN.

Commands are executed by ARIAN by simply looking up the command name in a table and doing a subroutine call to the associated address. This method was chosen to give modularity and ease of modification to

ARIAN. All arguments are passed in reserved buffer areas, thereby freeing the registers for general use. Both of these choices were made because of experience; they are considered to be the simplest and least-memory-consuming methods known to the author.

A unique feature of ARIES-I is its utility package, which is always system-resident in PROM. This 2K package, which supports over 50 utility subroutines, may be used by any program in the system. Interface is made to this package through a jump table. The idea of such a utility package is considered to be exceptionally useful since many programs can now run on ARIES-I without ARIAN being resident. A flexability is afforded which permits the user to work with very large operating systems which may require utility support and be forced to reside in the space occupied by ARIAN.

Another unique interfacing feature of ARIES-I is its integration with ARIAN to provide a non-maskable interrupt (supported by the Z80) feature. This interrupt can be executed regardless of the interrupt enable flip-flop in the Z80, and it provides a clean entry into ARIAN. In this way, control can be restored from an uncontrolled program and passed to ARIAN while preserving the local memory files. Note that this may not necessarily work if the uncontrolled program has destroyed a part of ARIAN or its files.

Thirdly, interfacing is done with ARIAN through the system subroutine ZINK. This subroutine, called from a user program, polls the

principal input port and returns control to ARIAN if an ESCAPE character was typed. This provides a clean return to ARIAN and gives the user extended control over his programs.

Fourth, interfacing can also be done through breakpoints. As mentioned in the users' manual, breakpoints can be set, and, if executed, they will return control to ARIAN and allow the user to examine the contents of the registers at the time the breakpoint was reached. This is considered to be an excellent debugging tool.

Program interfacing is handled in many ways by ARIAN and ARIES-I. Reserved labels, entry points, jump tables, hardware interrupts, and breakpoints are the major methods. These are all considered to be good interfacing methods and should be employed in future microcomputer operating systems.

I/O interfacing is another form of system interfacing supported by ARIAN. Through redirectable I/O (see the SETC command), the user may create his own I/O routines to be executed as principal I/O drivers for the system. To allow ease of interface with these routines, ARIAN passes the argument to be input or output in the A register and requires the user to write I/O routines which have no net effect on the any registers. I/O can be redirected to any device or set of devices at the user's discretion in this manner. This is a significant contribution of ARIAN.

# CHAPTER 7

## THESIS CONCLUSION

ARIAN and ARIES-I are integrated, specialized systems, and this type of configuration was how microprocessors were originally employed. This integration is complete; each system was designed with the other in mind.

Many lessons were learned during the implementation of Project ARIES. This thesis has served to summarize them. Further summary will not be presented now -- it would be redundant. The general conclusions, however, are of consequence and should be specifically outlined. These conclusions are:

1. Human engineering should be of prime concern in the development of a microcomputer operating system like ARIAN. Humans are to be constantly working with the operating system, and one of its goals should be to relieve the humans of as many problems in

the way of overhead as possible; they should be free to concentrate on the problems they are trying to solve. Problems like file management, program debugging, and system resource allocation should be left to the operating system. The author feels that ARIAN has succeeded in this goal.

2. The concepts employed in the file system are of consequence. The use of a simple file structure with built-in error detection facilities, the use of in-memory files to speed user interaction and increase throughput, the use of the disk only for mass storage of data, and the use of simple commands, such as EXEC, ASSM, and FILE, to manipulate the files and speed software development are just some of these concepts.

3. The major features of the assembler are also of consequence. These features include Z80 extensions to the INTEL-standard mnemonics, an extendable symbol table, the use of reserved system symbols to simplify the user's programming task and provide a number of utility subroutines, and the ability to write totally relocatable programs easily.

4. The ARIAN command structure is a fourth contribution of Project ARIES. The use of customized commands which allow the user to redefine system commands and create his own set of commands is a very valuable feature.

5. The incorporation of a system utility package with a jump table in PROM into ARIES-I is also of significance. Permitting user programs to execute without the main operating system being loaded and eliminating the need to constantly be rewriting common utility routines are just two of the advantages of the utility package.

6. Finally, all of the minor points learned during the implementation of Project ARIES are of significance. These include the concepts of redirectable I/O, program and system interface, program and data structure in the microcomputer environment, and a "feel" for the limitations of the microcomputer.

## REFERENCES

[1] Richard L. Conn; The Monitor Command System User's Manual; Technical Report, Number UIUCDCS-R-78-912, Department of Computer Science, University of Illinois; 1978.

[2] Robert W. Catlin; MUMS: A Modular Unified Microprocessor System; M.S. Thesis, Number UIUCDCS-R-76-809, Department of Computer Science, University of Illinois; 1976.

[3] Mostek Corporation; Z80 Programming Manual; Publication Number MK 78515; Published by Mostek Corporation, 1215 W. Crosby Rd., Carrollton, Texas; Copyright 1977.

[4] J.P. Tremblay and P.G. Sorenson; An Introduction to Data Structures with Applications; Published by McGraw-Hill, Inc., New York; Copyright 1976; BNF reference on pp. 71-81.

[5] Richard L. Conn; The Bootstrap Monitor System Users' Manual; ARIES-I Software Document; 1977.

[6] Richard L. Conn; os8080; lesson available on the PLATO computer system, University of Illinois; Copyright 1977 by Board of Trustees, University of Illinois.

[7] North Star Computers, Inc.; The North Star MONITOR, Version 1; North Star Computers, Inc., 2547 Ninth Street, Berkeley, California; Copyright 1977.

[8] Processor Technology Corporation; SOLOS/CUTER User's Manual; Processor Technology Corporation, 7100 Johnson Industrial Drive, Pleasanton, California; Copyright 1976.

[9] Software Technology Corporation; CUTER Monitor Program Source Listing; Software Technology Corporation, P.O. Box 5260, San Mateo, California; Copyright 1977.

[10] IMSAI Manufacturing Corporation; IMSAI 8080 Self-Contained System User's Manual, Revision 2; IMSAI Manufacturing Corporation; 1975.

[11] IMSAI Manufacturing Corporation; IMSAI 8080 Self-Contained System Source Listing, Revision 2; IMSAI Manufacturing Corporation; 1975.

[12] Richard L. Conn; ARIAN/ARIES-I Software Library, Volumes 1-4; ARIES-I Software Document; 1977-1978.

[13] Dick Wilcox; The Hobbyist's Operating System, Part 1; Kilobaud magazine, issue 1.

[14] George Lehmann; An Implementation of C for the INTEL 8080; M.S. Thesis, Report Number not available, Department of Computer Science, University of Illinois; 1978.

[15] Alfred C. Weaver; CS397 Lecture Notes; Department of Computer Science, University of Illinois; available in DCS library; Spring, 1977.

# APPENDIX I -- THE ARIAN USERS' MANUAL

# CHAPTER 1

## AN OVERVIEW OF ARIAN

ARIAN is an operating system, a program which allows its user to execute and control other programs, designed specifically for the ARIES-I microcomputer system. ARIAN is extensively integrated with ARIES-I; the microcomputer hardware and the ARIAN software are designed to work together with each other's configuration in mind. This type of design--that of hardware/firmware/software integration within a single system--produces a unique tool.

This tool is designed to be used for software development. The hardware configuration of ARIES-I, for example, was organized and coordinated with ARIAN to give the user the maximum amount of benefit from his resources. This hardware configuration includes the following:

1. 40K of RAM.

2. the Z80 microprocessor. This microprocessor was selected for ARIAN because it is compatable with the 8080 microprocessor, it provides a larger number of addressing modes and registers than the 8080, and its support circuitry (the VECTOR GRAPHIC MPU board) is based on the S-100 bus structure. The S-100 bus structure was chosen because this structure has by far the most support in the commercial world; the Z80 was chosen because of its enormous power, as seen in its wealth of addressing modes, its large number of registers, and its upward compatability with the 8080 microprocessor.

3. a floppy disk drive. This device is used for mass storage of programs and data.

4. a PROM programmer.

5. a high speed (9600 baud) CRT screen and keyboard, a printer (PORTACOM PC-8110), and a modem (PENNYWHISTLE 103). These devices, with their associated serial I/O ports, provide the I/O for ARIES-I. The CRT was chosen to minimize user discomfort while working with ARIES-I by maximizing I/O speed, the printer is necessary for hard copy, and the modem is necessary to communicate with an external computing system in Terminal mode (discussed later).

6. three parallel I/O ports. These I/O ports are provided to allow the user to interface additional hardware to ARIES-I and to pass additional information between the user and ARIES-I. Additional parallel I/O devices may be connected to ARIES-I through two of these ports, additional serial I/O devices may be connected to ARIES-I by disconnecting the modem or printer and using their ports, and ARIES-I provides a set of eight LEDS and eight switches on its front panel which the user may access through the third parallel port to transfer data for his specific applications. This front panel port is also used by ARIAN to indicate the high-order address of the downloaded code while in Terminal mode.

As the ARIAN user can see, the ARIES-I hardware configuration is designed specifically with software development in mind. This is also the case with the ARIES-I software configuration.

ARIAN is specifically designed to aid in software development for the Z80 microprocessor. Its 35 commands give the user the abilities to create and manipulate text and binary files, to assemble the text of an assembly language file, to execute and breakpoint his assembly language programs with a number of debugging aids, to communicate with an external computer over the modem, and to examine and modify the microcomputer's memory directly.

While ARIAN is running, the user may be in any one of a number of command or data entry modes. These modes, discussed in more detail later, are:

1. Terminal mode -- Terminal mode is an interactive subsystem of ARIAN. The user can communicate with an external computer via the modem and acoustic coupler while in this mode.

2. Block Line Entry mode -- This mode permits the entry of a block of lines into the primary file. It allows the user to enter lines into the primary file without typing the line numbers; ARIAN automatically prefixes each line with a line number and renumbers the primary file when the user exits this mode.

3. Command mode -- Command mode permits the user to type a command to ARIAN.

4. Edit mode -- This mode is the command system invoked by the EDIT command (see Chapter 2).

5. Utility mode -- This is the Utility Subsystem, invoked by the UTIL command (see Chapter 2).

6. Display mode -- Display mode is the mode in which ARIAN is displaying a directory or lines of text to the user. He can stop this display while it is being created by keying <esc>. All displays are paged, and the user is prompted with " Q?" at the bottom of each page to find out if he wants to continue the display; the user must respond with "N" to stop the display and anything else to continue.

This section of the thesis is a user's manual for ARIAN. Chapter 2 covers the ARIAN command set in detail, Chapter 3 tells the user how to use ARIAN, and Chapter 4 and Chapter 5 tell the user how to take advantage of some of the specialized commands in ARIAN such as the CUSTomize command and the ASSM (assemble) command. Finally, Chapter 6 describes the ARIES-I utility package, a comprehensive set of utility subroutines.

# CHAPTER 2

## THE ARIAN COMMAND SET IN DETAIL

ARIAN has an extensive set of commands which give the user a great deal of control over the microcomputer system. This control includes the abilities to: (1) examine and modify memory directly, (2) control the system environment and all the programs available to the user, (3) debug user programs through specialized system commands, and (4) assemble programs in ARIAN Z80 assembly language.

The purpose of this chapter is to catagorize the commands in ARIAN and explain them in detail. The following is a quick reference list of these commands. The arguments are expressed in String Description Language (SDL) notation.

1. System Control Commands

| | |
|---|---|
| CONT | CONT <hadr>? |
| CUST | CUST <cname> <hadr>? |

```
          CUSTD <cname>
          CUSTL
          CUSTN <cname>
          CUSTS
EXEC      EXEC  (<fname> !  <hadr>)?
          EXECB <fname>?
EXIT      EXIT
FDOS      FDOS
RESE      RESE
SETC      SETC
          SETCI <hadr>
          SETCO <hadr>
TABS      TABS
          TABSL
          TABSR
TERM      TERM
WORK      WORK  (<hadr> <hadr>)?
```

2. Primary File Editing Commands

```
APND      APND  <lnum>?
DEL       DEL   <lnum> <lnum>?
EDIT      EDIT  <lnum>?
FIND      FIND  <lnum>?
INS       INS   <lnum>
RNUM      RNUM  (<lnum> <inc>?)?
<lnum>    <lnum> <text>
```

3. Local File Control Commands

```
FCHK      FCHK  <fname>?
FILE      FILE  (<fname> <hadr>?)?
          FILEB <fname> (<hadr> <hadr>)?
LDEL      LDEL  <fname>
          LDELB <fname>
LDIR      LDIR
          LDIRB
LNAM      LNAM  <fname>
          LNAMB <fname>
LSCR      LSCR
          LSCRB
RCVR      RCVR  <fname> <hadr>
```

4. Disk File Control Commands

```
DDEL      DDEL  <fname>
DDIR      DDIR
          DDIRP
DNAM      DNAM  <fname>
```

```
TYPE    TYPE  <fname> <hadr>?
```

5. Local/Disk File Transfer Commands

```
LOAD    LOAD  <fname> <hadr>?
SAVE    SAVE  <fname>
        SAVEB <fname> (<hadr> <hadr> <hadr>?)?
```

6. List/Print Commands

```
LIST    LIST  (<lnum> <lnum>?)?
        LISTF (<lnum> <lnum>?)?
        LISTN (<lnum> <lnum>?)?
PRIN    PRIN  (<lnum> <lnum>?)?
        PRINF (<lnum> <lnum>?)?
        PRINN (<lnum> <lnum>?)?
```

7. Program Debugging Commands

```
BREK    BREK  <hadr>
        BREKD <hadr>
        BREKL
        BREKS
REGS    REGS  (<rname> (<val8> !  <val16>))?
```

8. Assembler Commands

```
ASSM    ASSM  <fname>?  (<hadr> <hadr>?)?
        ASSMF <fname>?  (<hadr> <hadr>?)?
        ASSML <fname>?  (<hadr> <hadr>?)?
        ASSMP <fname>?  (<hadr> <hadr>?)?
SYMT    SYMT
        SYMTS
```

9. Utility Command and Subcommands

```
UTIL    UTIL
              C <hadr> <hadr> <hadr>
              D <hadr>?  <hval>*
              E (<hadr> <hadr>?)?
              F <hadr> <hadr> <hval>*
              S <hadr>?
              X
```

The following is a String Description Language (SDL) representation of the general ARIAN command (ARIANCOMMAND):

```
<blank> : ' '
<digit> : '0' ! ... ! '9'
<hexdigit> : <digit> ! 'A' ! ... ! 'F'
<alpha> : 'A' ! ... ! 'Z'
<alphit> : <alpha> ! <digit>
CMNDNAME : <alpha> <alphit>*3
FILENAME : <alpha> <alphit>*7
LINENUMBER : <digit> <digit>*3
HEXVAL : <digit> <hexdigit>*3 ! '0' <hexdigit>*4
UTILITYSUBCOMMAND : <alpha> <blank>* (HEXVAL <blank>+)* HEXVAL?
<lnum> : LINENUMBER
<hadr> : HEXVAL
<hval> : '0' <hexdigit>*2 ! <digit> <hexdigit>
<cname> : CMNDNAME
<fname> : FILENAME
<val8> : <hval>
<val16> : HEXVAL
<rname> : "THE ZILOG-STANDARD MNEMONIC FOR ANY Z-80 REGISTER"
ARIANCOMMAND : CMNDNAME <alphit>* <blank>+ FILENAME? (<blank>+
(LINENUMBER ! HEXVAL) <blank>*)*3
```

As can be seen by the above description, a command in ARIAN can take a large number of forms. Examples of these forms are:

```
FILE PROGRAM
UTIL
LIST 100 2000
TEST 3000 0F0F0 40
REGS BC 40
SAVEB BINARY1 3400 34FF 06800
```

## System Control Commands

There are ten commands which give the user control over his program execution and the execution of ARIAN and the MCS and FDOS systems. These are the system control commands. Briefly, they are:

1. CONT -- continue from a breakpoint or execute with specific register values.
2. CUST -- create, delete, or rename a customized command; also, list or scratch all customized commands.
3. EXEC -- execute or assemble and execute a specified file or program.
4. EXIT -- exit to MCS.
5. FDOS -- transfer control to FDOS.
6. RESE -- reset ARIAN.
7. SETC -- set or reset a customized input or output driver.
8. TABS -- set, reset, or examine the tab settings.
9. TERM -- enter terminal mode.
10. WORK -- set or examine the workspace parameters.

### The System Control Commands

CONT CONT <hadr>?

The CONTinue command is one of two commands which allow the user to explicitly execute a program in the microcomputer's memory. CONT may be used to either continue a program's execution after a breakpoint has been encountered or explicitly execute a program at a specified address.

When a breakpoint is encountered, the contents of all the registers of the microprocessor are saved in the register save area in the ARIAN scratchpad RAM, thereby permitting the user to examine and modify these values at his leasure through the REGS command. CONT with no argument will permit the user to continue execution of the program under test with the registers of the microprocessor loaded with the values stored in the register save area.

CONT with a specified address as an argument also loads the registers of the microprocessor, but execution is begun at the address specified. Hence, dynamic debugging of user subroutines is possible through this command.

Example: CONT 4000

Result: The values stored in the register save area are loaded into the appropriate registers and the program starting at location 4000 hexadecimal is executed.

| | | |
|---|---|---|
| CUST | CUST | <cname> <hadr>? |
| | CUSTD | <cname> |
| | CUSTL | |
| | CUSTN | <cname> |
| | CUSTS | |

CUSTomize is perhaps one of the most powerful commands in ARIAN's repertoire. This command allows the user to add his own set of commands to those already executed by ARIAN. The user may give his additional commands any names he wishes, including the names of the commands already defined by ARIAN. If the user redefines an ARIAN command in this manner, his customized subroutine replaces the system subroutine normally used to execute that command. This replacement is in effect until the user resets ARIAN or deletes his customized command. Other options under the CUST core include CUSTD to delete a customized command, CUSTL to list all the customized commands and their execution addresses, CUSTN to rename a customized command, and CUSTS to scratch (delete) all the customized commands.

The CUST command by itself is used to create or redefine a customized command. CUST with no address creates a customized command which will begin execution at the default address; CUST with an address defines the command explicitly to execute at the specified location. If a customized command of the same name already exists, the user is prompted with the "REPLACE?" message, to which he responds with a "Y" if he wishes to redefine the execution address of that customized command or "N" if he does not.

The other CUST commands require no detailed explanation; CUSTD deletes the specified command from the customized command table, CUSTN allows the user to rename a specified customized command, and CUSTL and CUSTS list and scratch, resp., all the defined customized commands.

Example: CUST TEST 4000

Result: The customized command named TEST is created. Whenever TEST is typed while in ARIAN command mode, the program at location 4000 hexadecimal will be executed.

Example: CUSTD TEST

Result: The TEST command created in the previous example is deleted from the customized command directory. From this point forward, whenever the user types the TEST command, ARIAN will respond with "$ INVALID COMMAND" until the TEST command is redefined.

EXEC  EXEC  (<fname> ! <hadr>)?
      EXECB <fname>?

The EXECute command is one of the most complex of the ARIAN commands. EXEC allows the user to execute a text file, a binary file, or any program or subroutine which resides in memory. This command also permits a clean return to ARIAN by pushing a return address onto the system stack (which may also be used as the program stack); by keeping the stack stable, the user may return to ARIAN when his program is finished by simply executing a return.

The EXEC command works in many implicit modes. EXEC with no arguments will assemble and execute the primary file. This means of program execution makes software development somewhat easier by permitting the user to execute his primary file, interrupt the program if it malfunctions, edit the file, and reexecute it with a minimum of effort.

EXEC <fname> will perform the following operations in the order specified:

1. It will first search the local file directory for the specified file. If this file is found, it will be made primary, assembled, and executed.
2. Secondly, if step 1 fails, it will search the disk directory for the specified file. If the file is found, it will be loaded into memory.
3. If the loaded file is a binary file, it will be immediately executed; if it is a text file, it will be assembled and executed.

4. If no file with the specified name is found, an appropriate error message will be given.

5. In all cases except when an error occurs, a binary file of the specified name will be created.

If EXEC <hadr> is used, the binary program starting at the specified address will be immediately executed.

Finally, if EXECB is used, the specified binary file (as defined in the binary file directory) will be executed; if no file name is specified, execution will begin at the default address.

Example: EXEC PROGRAM

Result: If PROGRAM is a local text file, it will be assembled and executed. If PROGRAM is not local and resides on disk, it will be loaded, assembled if it is a text file, and executed.

Example: EXEC 0F000

Result: The machine code beginning at location 0F000 hexadecimal is executed immediately.

Example: EXECB PROGRAMB

Result: The binary file PROGRAMB is executed immediately.

EXIT EXIT

The EXIT command simply transfers control to the PROM-resident Monitor Command System (MCS). Refer to the MCS User's Manual [1] for a detailed description of MCS.

Example: EXIT

Result: MCS is executed immediately. The system status of ARIAN is not affected unless the user explicitly changes ARIAN or its system RAM area through MCS.

FDOS FDOS

FDOS transfers control to the FDOS disk bootstrap program. FDOS is a modified version of Northstar Corporation's DOS program.

Example: FDOS

Result: Control is transferred to FDOS. Again, the system status of ARIAN is not necessarily affected.

RESE RESE

RESEt executes a system reset of ARIAN. This system reset includes the following operations:

1. The default execution address is set to the system reset entry point.
2. The assembly limits are reset.
3. The system symbol table is cleared.
4. The system tab stops are reset.
5. The default stack pointer is reset for the CONT command.
6. The default disk drive number is reset to 1.
7. The serial I/O port is reset.

8. Control is returned to ARIAN.

Example: RESET

Result: ARIAN is reset.

SETC

```
SETC
SETCI   <hadr>
SETCO   <hadr>
```

The SETC command allows the user to redirect all ARIAN system I/O through user-defined I/O routines. The parameter to be passed to and from these routines is passed in the A register. The only constraint placed on these routines is that they do not have a net effect on the system stack or any register and they end in a return instruction.

SETC by itself resets the system I/O to employ the normal system I/O routines. This is the default upon ARIAN initialization.

SETCI (set customized input) allows the user to redefine the system input routine. All input is routed through the subroutine which begins at the specified address immediately after this command is executed.

SETCO (set customized output) allows the user to redefine the system output routine. All output is routed through the subroutine which begins at the specified address immediately after this command is executed.

Example: SETCI 6C00

Result: All input to ARIAN is directed through the subroutine starting at location 6C00 hexadecimal until the user resets the I/O routine through a system reset (RESE) or SETC.

TABS TABS
TABSL
TABSR

The TABSet command allows the user to set, examine, and reset the tab stops. In the set and examine cases, a scale numbering the columns is printed across the CRT screen, and the tab stops are denoted by the letter "X".

TABS is used to set the tab stops. After the scale is printed, the user may space or tab (using the previously-defined tab stops) over to the desired tab set location and type an "X". Up to twenty tabs may be

set in this way; the process is terminated by typing a carriage return.

TABSL examines (lists) the tab stops as they are currently set. Again, the scale is printed and X's are printed in the columns in which tabs are set.

TABSR resets the tab stops to the standard system definition. Tabs are set at every eighth column.

Example: TABSL

Result: A column scale is printed across the user's CRT screen and the settings of the current tab stops are indicated by X's under the appropriate column numbers.

TERM TERM

The TERMinal command invokes the terminal, or timesharing, mode of ARIAN. Under this subsystem, the microcomputer becomes relatively transparent to the user and the user's terminal is made to respond like a normal timesharing terminal to an external computer system. Each character typed is sent to a modem and acoustic coupler, and each character received by the modem/coupler is sent to the CRT screen.

The terminal subsystem responds to three subcommands; they are ctrl-A, ctrl-L, and ctrl-R. Ctrl-A invokes the terminal alternate command set; ctrl-L invokes the download program; ctrl-R returns control to ARIAN.

The terminal alternate command set consists of the following MCS-like commands:

1. M -- return to MCS.
2. R <hadr> -- run the subroutine located at the specified address. The return in the subroutine transfers control to the terminal mode (not the terminal alternate command set).
3. T <hadr> -- transfer the downloaded code to the specified address. This command transmits a carriage return to the external computer, and the object code downloaded is loaded into memory starting at the specified address; the addresses given in the code are ignored, and the entire code is loaded sequentially into memory.
4. X -- exit to terminal mode.

Ctrl-L is the download command to the terminal mode subsystem. A carriage return is transmitted to the external computer, and the object code, in an ARIES-I format, is loaded into the microcomputer's memory at the addresses specified in the load blocks.

Ctrl-R simply returns control to the program that called the terminal subsystem. This calling program is usually ARIAN.

Example: TERM

Result: The TERMinal subsystem is invoked.

Example: TYPE,OBJECT ^L

Result: This is the TYPE command to the external computer. OBJECT is an object code file in one of the ARIES-I formats. When ARIES-I reads the ctrl-L from the user, it transmits a <cr> to the external computer, thereby terminating the TYPE command. ARIES-I then downloads the incoming code into memory and displays it on the CRT screen.

WORK WORK (<hadr> <hadr>)?

The WORKspace command is used to display and set the local text file workspace boundaries. These boundaries are used by the local memory manager and the assembler to determine where to place new files and where to move old files when a local file manipulation is required, to monitor the growth of the primary file, and to determine where to place assembly code when the default address is to be used.

All local text files are forced to reside within the defined workspace unless the user explicitly overrides the memory manager. The default assembly address is set to be at the byte immediately following the workspace. ARIAN defines the workspace to fill all but 2K bytes of the contiguous memory block immediately following FDOS.

WORK displays the workspace boundaries; WORK with two addreses redefines these boundaries to the addresses specified.

Example: WORK

Result: The current workspace boundaries are displayed on the user's CRT.

Example: WORK 3000 5000

Result: The user's workspace boundaries are reset to include locations 3000 to 5000 hexadecimal, inclusive. Therefore, the default assembly address is 5001 hexadecimal.

## Primary File Editing Commands

There are seven commands which give the user the ability to edit and manipulate the primary file. Briefly, the primary file editing commands are:

1. APND -- place the following block of lines after the specified line in the file. The block of lines is then typed in by the user in unnumbered mode.

2. DEL -- delete the specified line or lines from the primary file.

3. EDIT -- invoke the intra-line editor on the specified line of the primary file.

4. FIND -- find all occurrances of the specified string in the primary file.

5. INS -- place the following block of lines in front of the specified line in the file. The block of lines is then typed in by the user in unnumbered mode.

6. RNUM -- renumber the primary file.

7. <lnum> -- enter the line beginning with this number into the appropriate place in the primary file. If a line already exists with this number, it is replaced; otherwise, an insertion is done.

APND APND <lnum>?

The APND (append) command is one of two block line entry commands of the ARIAN editing system. Block entry in ARIAN permits the user to type an arbitrary number of lines into the primary file without typing their corresponding line numbers. When the user has finished typing this group of lines, he then types a ctrl-C followed by a carriage return. These lines will then be entered into the primary file at the appropriate place and the entire file will be renumbered. If the ctrl-C is at the end of a line, it will not be included in this line, and this line will be the last line of the block; if ctrl-C is the first character of a line, the previous line will be the last line of the block.

APND without a line number will enter the block of lines after the last line in the file. APND with a line number will enter the block of lines after the specified line and before the next line in the file.

Example: APND

Result: The user is prompted with a "?", after which he types a line of text. If this line is not terminated by a ctrl-C and a <cr>, another prompt appears and he may then type another line of text. This process continues until he either ends a line with a ctrl-C <cr> or

begins a line with a ctrl-C <cr>. At this time, the block of lines he has just typed will be appended to the primary file and the primary file will be renumbered.

Example: APND 100

Result: The same procedure is executed as described above, but the block of lines is inserted between line 100 and the next line in the file.

DEL DEL <lnum> <lnum>?

The function of the DELete command is obvious: it deletes lines from the primary file. DEL followed by a line number will delete only that line; DEL followed by two line numbers will delete the block of lines enclosed by the specified lines, inclusive. If a specified line number does not exist in the file, the line number of the line which would follow the specified line if it exists will be used; if the specified line number is larger than the largest line number in the file, the last line will be deleted.

Example: DEL 100

Result: Line 100 is deleted from the primary file.

Example: DEL 200 235

Result: Lines 200 to 235, inclusive, are deleted from the primary file.

EDIT EDIT <lnum>?

The EDIT command invokes the ARIAN intra-line editor. This editor allows the user to edit a line that has already been typed without retyping the entire line. If a line number is not specified, the first line of the file will be edited; if a line number is specified, that line, if it exists, or the line that would follow it if it did exist, will be edited.

The intra-line editor is a dynamic editor which permits the user to see the effects of his editing commands immediately after he types them. When a line is edited, it is copied into the editor's old line buffer and then displayed to the user. The editor then does a carriage return

and prompts the user with a question mark. As the user edits this line, each character of the new line that is created is placed into the editor's new line buffer; the original line in the old line buffer is not affected. Finally, when editing is finished, the user may type a carriage return to terminate the editing process and replace the original line in the file with the line as it exists in the new line buffer.

The intra-line editor responds to a host of subcommands. The following is a complete list of these commands and their functions.

1. <sp> -- copy the character pointed to by the old line pointer into the character position pointed to by the new line pointer and advance the old line and new line pointers by one. The space bar, therefore, will simply copy the next character from the old line buffer into the new line buffer. After the copy is done, the copied character will be displayed to the user.

2. E -- skip to the end of the line. The rest of the characters in the old line buffer are copied into the new line buffer and both pointers are advanced to point to the non-existent character after the last character copied. The copied characters are displayed to the user as they are copied.

3. D -- delete the character pointed to by the old line pointer (delete the next character in the old line). The character is deleted by advancing the old line pointer by one character position and not affecting the new line pointer. The deletion is displayed to the user as a backslash ("\") followed by the deleted character. If the next command typed by the user is another D, the next deleted character is displayed (without the backslash). This will continue until the user types some other command, in which case a closing backslash will be printed and that command will be executed. In effect, the deleted characters are enclosed in backslashes when displayed to the user.

4. I -- insert a string of characters in front of the character currently pointed to by the old line pointer. In response to the I typed by the user, the editor types a slash ("/"). The user may then type any string of characters he wishes except

for an escape or a carriage return. These characters will be copied into the new line buffer, the new line pointer will be advanced, and each character will be echoed to the user as he types it.

The escape and carriage return characters are special characters to the insert subcommand. <esc> instructs the insert subcommand to end the insertion. The editor then types another slash to indicate that the insertion is finished and allows the user to continue editing normally. <cr> instructs the editor to terminate creation of the new line, copy the new line into the primary file, and return to ARIAN command mode. The <cr> is echoed as a slash, a carriage return, and a system prompt (">"), indicating that the user is now in ARIAN command mode.

5. R -- replace the characters pointed to the the old line pointer with the following string. Both pointers are advanced and the new characters are echoed to the user. No special character is typed to the user after he types an R, and the <esc> and <cr> characters respond as in the I command except that no slash is printed. In effect, as the user types his string, each character he types replaces the corresponding character in the old line buffer.

6. S <letter> -- skip to the specified letter. This is the only two-character command in the editor; it consists of the letter S followed by a single character. When this command is typed, both the old and new line pointers are advanced and the corresponding characters are typed and copied into the new line buffer until the specified character is encountered or the end of the line is reached. Once the specified character is found, the old line pointer will point to it and this character will not be printed; it will be the next character in the line. The S and the specified letter are not echoed to the user when the command is typed. This command is very useful, particularly when the user wishes to insert, delete, or replace at a specified character; he does not have to space over to that character with this command.

7. <del> -- the delete key backs up the new line pointer. The characters backed over are enclosed in "<" and ">" (like they are enclosed in slashes in the I command) and deleted from the new line. Only the new line pointer is affected by this command.

8. <cr> -- terminate creation of the new line. This command terminates editing of the line and replaces the original line in the primary file with the line that currently exists in the new line buffer. If <cr> is the first editing character typed, the edit is aborted and no replacement occurs.

9. A -- abort the editing of the old line. This command may be typed whenever the editor is ready to receive a command (i.e., the editor is not in the middle of an insertion or replacement), and it terminates the edit and returns control to ARIAN without affecting the original line.

10. P -- print the new line and edit it. This command will terminate the new line at the current position of the new line pointer, copy the new line buffer into the old line buffer, print the new line, and restart the editing sequence with this new line instead of the original line. The original line as it exists in the primary file is not affected.

11. X -- exit and reedit the old line. The X command terminates the editing done so far and restarts the edit of the original line. If a P command has been previously typed, the last line placed into the old line buffer is reedited.

The editor has three error messages that it may display. These messages are:

1. ?? -- invalid command. A double question mark indicates that an invalid command has been typed.

2. ** -- end of edit line. A double asterisk indicates that the user has tried to go beyond the end of the original line illegally while editing. This error most commonly occurs while in the R command, and the user should respond to this error by typing <esc> followed by the I command to continue entering the characters.

3. *EOL* -- end of line buffer. The length of the new line has just reached the limits of the new line buffer, and the user must reedit the original line.

Example: EDIT 200

Result: Line 200 is printed and the user is prompted with a "?". The user may now edit line 200 using the intra-line editing commands. One useful aspect of this command not yet discussed is that line 200 may be copied as line 201 or any other desired line number by editing only the line number (like changing the second zero in 200 to a 1) and typing the E (skip to end of line) command followed by a <cr>. Line 200 in the primary file will be unchanged and line 201 will be created; line 201 will be a copy of line 200.

FIND FIND <lnum>?

The find command performs a search through the primary file for a string specified by the user. In response to this command, ARIAN will print "SEARCH STRING?", to which the user may respond by typing the string he wishes to search for followed by a <cr>. If the user responds with simply a <cr>, the search is aborted and control is returned to ARIAN.

If no line number is specified, the search is done over the entire file; if a line number is specified, the search is done starting at the specified line and extending to the end of the file.

Example: FIND

Result: ARIAN will respond with the phrase "SEARCH STRING?", to which the user will type the string he wishes to search for, like "ORA A", for example. ARIAN will then print in paged mode all the lines in the primary file containing this string.

INS INS <lnum>

The INSert command is the same as the APND command, except that the INS command places the block of lines in front of the specified line while APND places the block after the specified line. INS must have a line number, and it is particularly useful in inserting lines before the first line in the file. APND, on the other hand, is useful for appending lines to the end of the file.

Example: INS 300

Result: The same block line entry will occur as in the APND command described above, but the block of lines will be inserted between the line preceeding line 300 and line 300.

RNUM RNUM (<lnum> <inc>?)?

RNUM renumbers the primary file. If no arguments are given, the file is numbered to start at line 10 and continue in increments of 10. If just a line number is specified, the file starts at the specified line number and continues in increments of 10; if both line number and increment are specified, these values are used in the renumbering.

Example: RNUM 100 40

Result: The primary file will be renumbered, starting with line 100 and incrementing by 40; i.e., the first line will be line 100, the second line 140, the third line 180, etc.

Example: RNUM

Result: The primary file will be renumbered, starting with line 10 and incrementing by 10.

<lnum> <lnum> <text>

The user may insert a line of text into the primary file or replace a line already in the primary file by simply typing the desired line number followed by a space and the text of the line. This feature of ARIAN provides for very simple line insertion and replacement.

Example: 325 THIS IS LINE 325

Result: The above line (325) is entered into the primary file. If line 325 already exists, it will be replaced by the above line; if it does not exist, line 325 will be inserted between lines 324 and 326 (or the nearest lines to this range) in the primary file.

## Local File Control Commands

The local file control commands give the user explicit control over the files in the local text and binary file directories. This control includes the abilities to check the validity of a text file, define and create text and binary files, delete files from the directories, rename files in the directories, scratch (delete) all directory entries, display the directories, and recover a text file that has been deleted from the text file directory. These commands include:

1. FCHK -- check the validity of a local text file. This command checks the specified or implied file as to its correctness in format.
2. FILE -- create/display a primary file.
3. LDEL -- delete the local file specified.
4. LDIR -- display the local file directory specified.
5. LNAM -- rename the specified local file.
6. LSCR -- delete the specified directory.
7. RCVR -- recover a deleted file and make it primary.

## Local File Control Commands

### FCHK FCHK <fname>?

The file check (FCHK) command is used to determine the validity of the specified local text file. This verification includes checking to see that the character count for each line is correct, each line terminates with a <cr>, the file is properly terminated by an <eof> mark, and the directory limits for the file are correct. ARIAN will respond with "VALID FILE" or "INVALID FILE" when the test is finished.

The FILE and RCVR commands do an implicit file check whenever they are executed, but only the invalid message is displayed by their implicit checks.

Example: FCHK

Result: The current primary file is checked for validity.

Example: FCHK SECOND1

Result: The secondary file SECOND1 is checked for validity. SECOND1 is not made primary.

FILE

FILE (<fname> <hadr>?)?
FILEB <fname> (<hadr> <hadr>)?

The FILE command is one of the most powerful and complex commands in ARIAN. This command is used to create the primary file or a binary file and display the directory entry for the primary file.

The FILE command with no arguments displays the directory entry for the primary file. This entry, like the entries for the secondary local files, consists of the name of the file and the starting and ending memory addresses of the file.

The FILE <fname> variant creates a primary file of the name specified. If a local file already exists with this name, it is made primary; the memory manager is invoked, the new primary file is moved to the physical end of the old primary file, and the files are compacted within the currently-defined workspace boundaries. If no local file with this name exists, the new primary file of length zero is created at the end of the last primary file and compaction is again done. Text may

now be entered into the primary file by typing line numbers or using the APND command.

If an address is specified in the FILE <fname> variant, the new primary file is placed at this address. Compaction is done to the secondary files, but the new primary file is unaffected by this compaction. Also, the workspace boundary parameters are ignored when the primary file is placed, permitting the user to place this file anywhere he wishes. This FILE variant, then, effectively overrides the memory manager; however, if a later command uses the memory manager, the primary file may be moved and compacted by the memory manager implicitly.

FILEB permits the user to explicitly create a binary file. Binary files are defined solely by their entries in the local binary file directory; they conform to no particular physical structure as they exist in memory. FILEB with no numeric argument creates a binary file with the specified name at the default binary file limits set by the last assembly; FILEB with the two addresses creates the binary file with these addresses as its boundaries.

Example: FILE NEW

Result: If file NEW already exists locally, it is made primary; otherwise, a new text file is created with the name "NEW" at a location determined by the workspace manager. File compaction and workspace compression is then done to all local files.

Example: FILE

Result: The local directory entry is displayed for the current primary file. If this follows the above example and NEW did not previously exist, then an entry like

NEW 3020 3020

would be displayed to the user. This indicates that the current primary file is named NEW, it is a null file, and it begins and ends at location 3020 hexadecimal. If NEW is not null, an entry like

NEW 3020 304C

would be displayed to the user. In this case, NEW resides at locations 3020 to 304C hexadecimal, inclusive.

Example: FILEB BINARY 5000 5010

Result: The local binary file directory entry for the file named BINARY is created. This command defines the file BINARY to reside

at locations 5000 to 5010, inclusive.

LDEL LDEL <fname>
LDELB <fname>

The local file delete command (LDEL) is used to delete the directory entry of the specified local binary or text file. This command only deletes the directory entry; the physical file is unaffected by it. LDEL deletes the specified text file, while LDELB deletes the specified binary file.

Example: LDEL NEW

Result: The local text file directory entry for the file NEW is deleted. File NEW itself is unchanged and may be recovered.

Example: LDELB BINARY

Result: The binary file directory entry for the file BINARY is deleted. The binary file is unaffected by this command.

LDIR LDIR
LDIRB

The local directory command displays the local text and binary file directories. LDIR displays the local text file directory and LDIRB displays the local binary file directory. All directory entries consist of the names of the files and their current memory address boundaries. LDIR will display the entry for the current primary file first, and this entry is followed by the entries for the secondary files.

Example: LDIR

Result: A listing of all the current local text files is displayed to the user. A listing of files may look like:

```
F1       2100 2250
MYFILE   2001 20FF
SEMINUL  2000 2000
```

The file F1, residing at 2100 to 2250, inclusive, is the primary file and MYFILE and SEMINUL are secondary files. Note that the primary file is located after the last secondary file.

Example: LDIRB

Result: The local binary file directory is displayed. Such a display may look like:

```
6800 683F
F1 6800 683F
```

This display indicates that the addresses covered by the last assembly were 6800 to 683F hexadecimal, inclusive, and that a binary file named F1 (not the same as the text file F1 necessarily) exists at 6800 to 683F. 6800 is the default execution address.

LNAM LNAM <fname>
LNAMB <fname>

The local rename (LNAM) command allows the user to rename any local file. LNAM renames a local text file, and LNAMB renames a local binary file. In response to this command, ARIAN prompts the user with "NEW NAME?", to which he may respond with the desired new name for the specified file or a <cr> to abort the renaming process.

Example: LNAM F1

Result: ARIAN responds with the prompt "NEW NAME?", to which the user may respond with a valid file name or a <cr>. If he responds with a file name, like FILE1, F1 is renamed FILE1 in the local text file directory; if he responds with a <cr>, the renaming is aborted.

LSCR LSCR
LSCRB

LSCR scratches (deletes) the specified local file directory. The specified directory is initialized, and all entries are erased. LSCR scratches the local text file directory and LSCRB scratches the local binary file directory.

Example: LSCR

Result: The local binary text file directory is scratched. If the user next types the LDIR command, nothing will be printed and control will return to ARIAN.

RCVR RCVR <fname> <hadr>

The recover (RCVR) command is used to recover a text file that has been deleted from the local text file directory. A validity check is done on the file (see FCHK) starting at the address specified, and a local text file directory entry is created with the specified file name and the file boundaries ascertained by the validity check. If the validity check fails, the recovery is aborted with an appropriate error message. If the validity check succeeds, the new file is made primary implicitly through the FILE command.

Example: RCVR FILE 2001

Result: The deleted file, previously known as NEW, is recovered under the name of FILE if it is still valid. If the user now types the LDIR command, the following will be displayed:

```
FILE 2001 20FF
```

## Disk File Control Commands

This is a small set of commands which permit the user to obtain a directory of the files contained on disk, delete a disk file, and rename a disk file. These commands do not affect the local files. Briefly, the disk file control commands are:

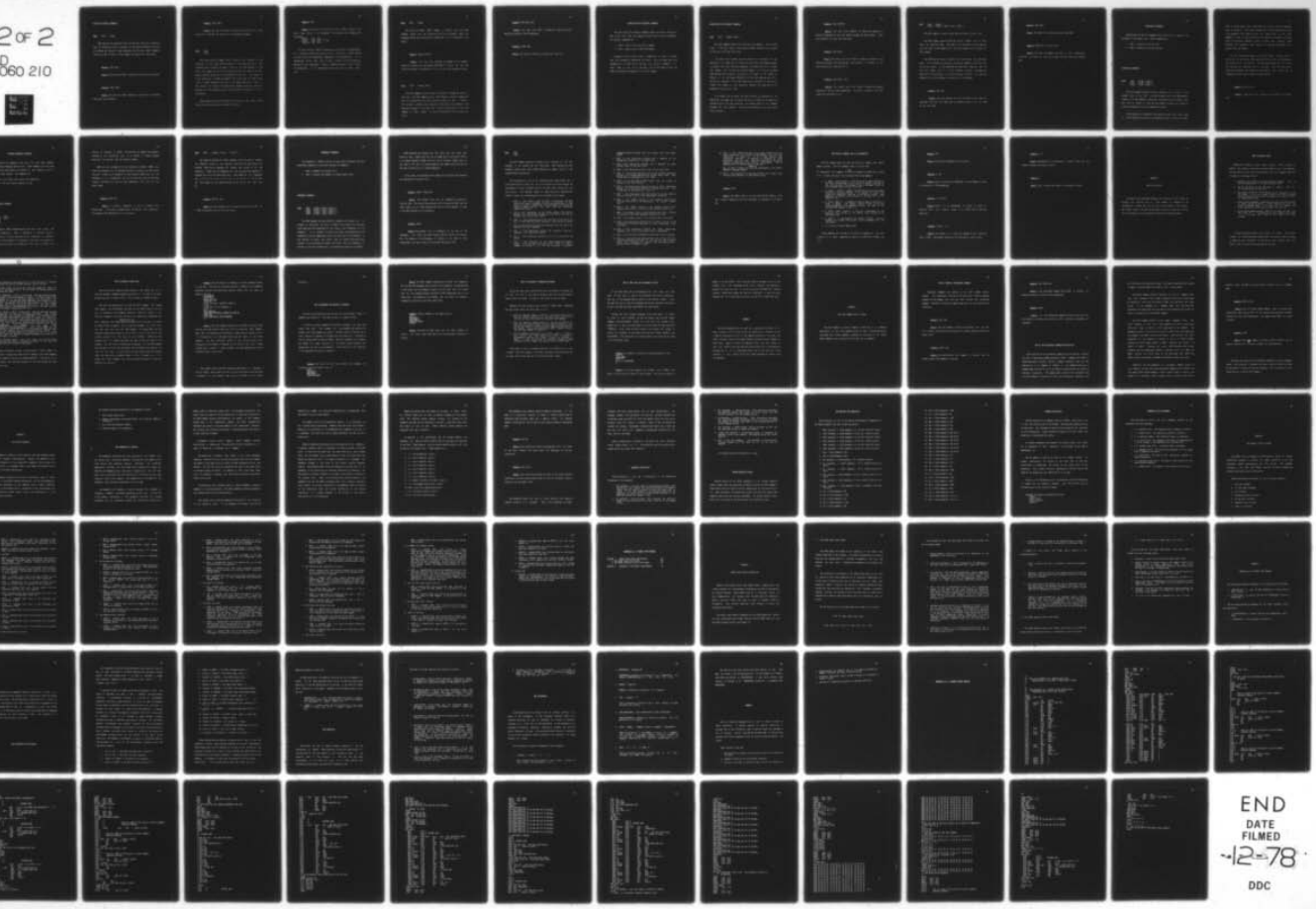
1. DDEL -- delete the specified disk file.
2. DDIR -- display a directory of the files on disk.
3. DNAM -- rename the specified disk file.
4. TYPE -- change the type or execution address of a file

All disk file control commands may be followed by an optional digit from 1 to 3 (like, DDIR3, DNAM1). This digit is used to specify the disk drive which the command will use. After a drive number has been specified, all disk file control commands will use that drive until another drive is specified. If the microcomputer has only one disk drive, the digits will be ignored. ARIAN initializes the default drive number to 1. Hence, DDIR after ARIAN startup gives a directory of drive 1.

AD-A060 210 ARMY MILITARY PERSONNEL CENTER ALEXANDRIA VA F/G 9/2
ARIAN -- AN IMPLEMENTATION OF A MICROCOMPUTER OPERATING SYSTEM.(U)
MAY 78 R L CONN

UNCLASSIFIED NL

2 OF 2
AD
A060 210



END
DATE
FILMED
12-78
DDC

## Disk File Control Commands

**DDEL** DDEL <fname>

DDEL deletes the specified disk file from the disk file directory. Only the directory entry is deleted, but the space occupied by the file is released and subject to user compaction and the disk space manager. Unlike for LDEL or LSCR, no RCVR command is supplied for disk files.

Example: DDEL TEXT1

Result: The disk file TEXT1 is deleted from the disk directory.

Example: DDEL3 TEXT1

Result: The disk file TEXT1 residing on disk drive 3 is deleted from that disk directory.

Example: DDEL TEXT2

Result: The disk file TEXT2 is deleted from disk drive 3. Note that drive 3 was made the default by the preceeding command.

DDIR DDIR
DDIRP

The disk directory command (DDIR) displays the contents of the directory of the disk currently mounted in the specified drive. Each directory entry contains the name of the file, the disk address of the file, the length of the file in 256-byte blocks, the type of the file, and an optional execution or load address for the file. The types of files permitted in ARIAN are general text files (type 0), binary files (type 1), BASIC program files (type 2), and BASIC data files (type 3). The entries for binary files and BASIC program files also contain an execution or load address at which the file must be loaded if it is to be executed.

DDIR displays the disk directory on the user's CRT, while DDIRP prints the disk directory on the user's printer.

Example: DDIR

Result: A directory of the files on the current default disk drive (disk drive 3) is displayed. The directory will resemble the following:

```
MYFILE1   0034  0002  0
FILEX     0036  001F  1  30F0
LASTFILE  009F  0020  0
```

In this directory, MYFILE1 resides at disk location 34 hexadecimal, is 2 256-byte blocks long, and is a type 0 (text) file. FILEX resides at disk location 36 hexadecimal (immediately following MYFILE1), is 1F hexadecimal blocks long, and is a type 1 (binary) file with execution address at 30F0 hexadecimal. Finally, LASTFILE resides at disk location 9F hexadecimal, is 20 hexadecimal bytes long, and is a type 0 (text) file.

DNAM DNAM <fname>

The disk file rename (DNAM) command is exactly like the LNAM command, except that the specified disk file is renamed. Again, the renaming process may be aborted by typing just a <cr> in response to the prompt.

Example: DNAME LASTFILE

Result: The disk file LASTFILE is renamed by this command. ARIAN will respond with the phrase "NEW NAME?", to which the user may type the new name for LASTFILE or a <cr> to abort the renaming process.

TYPE TYPE <fname> <hadr>?

The TYPE command allows the user to explicitly change the type of a disk file. The TYPE command with no fifth special character changes the type of the specified file from its current type to type 1 (binary). The optional address must be given in this case; this address is the execution address of the binary file. The only other form of the TYPE command is TYPE0 <fname>, in which the specified file is typed as 0 (text).

Example: TYPE TEST F000

Result: The disk file TEST is typed as a binary file with execution address at F000 hexadecimal.

Example: TYPE0 TEST

Result: The disk file TEST is now typed as a text file.

## Local/Disk File Transfer Commands

The local/disk file transfer commands permit the user to explicitly load files from disk into memory and save files on disk from memory. The transfer commands are:

1. LOAD -- load a file from disk to memory.
2. SAVE -- save a file on disk from memory.

Whenever a load or save is done, a comparison is done to ensure that the information transferred is correct. Also, the same disk drive designation is in effect as in the disk file control commands; i.e., LOAD1 loads on disk drive 1 and sets 1 as the new default drive, and SAVE3 and SAVEB2 are examples for the SAVE command.

## Local/Disk File Transfer Commands

### LOAD  LOAD <fname> <hadr>?

The LOAD command loads a file from disk into memory. Only binary (type 1) and text (type 0) files may be loaded; an error will be given if the file is of some other type.

If a text file is loaded, the disk directory is searched for the specified file name, and, if found, the local text file memory manager is invoked, the local files are compacted, the specified file is loaded into memory, and the new file is made primary. If a file by the same name already exists locally, the user will be asked if he wishes to replace it by the prompt "REPLACE?"; if the user responds with a "Y", the file is replaced. If the user specifies an address, the new file is loaded into memory at the specified address, but compaction of the secondary files is still done.

If a binary file is loaded, the disk directory is searched for the specified file name, and, if found, the file is loaded at the execution address given by the disk directory. Any address given in the command becomes the load address. An entry is then made in the local binary file directory.

Example: LOAD LASTFILE

Result: The disk file LASTFILE is loaded into memory at a location designated by the local memory manager and made primary. Note that LASTFILE is a text file from above.

Example: LOAD FILEX

Result: The binary disk file FILEX is loaded into memory at its execution address (30F0 hexadecimal), and an entry is created in the local binary file directory for it.

Example: LOAD FILEX1 40F0

Result: The binary disk file FILEX1 is loaded into memory starting at location 40F0 hexadecimal. An entry is created in the local binary file directory for it.

**SAVE**

```
SAVE    <fname>
SAVEB   <fname> (<hadr) <hadr> <hadr>?)?
```

The SAVE command is used to save text and binary files on disk.

The SAVE <fname> variant saves the current primary file on disk under the specified name. This name is not required to be the same as the local name of the primary file. Only the primary file is saved by this command.

The SAVEB variant saves a binary file on disk under the specified name. If no address is specified, the default assembly limits are used as the file limits. If two addresses are specified, these are used as the file limits, and if a third address is given, it becomes the execution or load address. If no third address is given, the starting address of the file becomes its execution or load address.

Example: SAVE NEW

Result: The local primary text file is saved on disk under the name NEW. Note that this name does not necssarily have to be the same as its local name.

Example: SAVE NEW2

Result: The same file is saved under the name NEW2.

Example: SAVEB BIN1 2000 2021 5600

Result: The section of memory from 2000 to 2021 hexadecimal, inclusive, is saved on disk as a binary file with execution address 5600.

## List/Print Commands

ARIAN gives the user two commands which allow him to display the contents of the primary file. These commands are:

1. LIST -- display on the user CRT.
2. PRIN -- display on the user printer.

List/Print Commands

LIST

```
LIST    (<lnum> <lnum>?)?
LISTF   (<lnum> <lnum>?)?
LISTN   (<lnum> <lnum>?)?
```

The LIST command allows the user to display all or part of the primary file on his CRT. All LIST variants are of the same general format: if no line number is specified, the entire file is listed, only one line is listed if just one line number is given, and a block of lines is displayed if two line numbers are given.

LIST displays the requested lines exactly as the user typed them in. LISTF displays the lines in an assembler format in which all labels

start in column eight, all op codes start in column 16, and all operands start in column 24. This format assumes that the user entered his lines in an assembler free format in which all labels start in the first assembly column of each line and all op codes not proceeded by a label start in the second assembly column of each line. Finally, LISTN displays the requested lines exactly as the user typed them in but without the line numbers.

All list displays are paged 15 lines per screen. If more than 15 lines are to be listed, ARIAN pauses after each 15th line and prompts the user with "Q?" to continue. The user must respond with "N" to stop the listing and anything else to continue. Also, the <esc> key is monitored throughout the creation of the display, and listing can be terminated at any time by simply hitting this key.

Example: LIST 300 400

Result: Lines 300 to 400, inclusive, are listed on the user's CRT.

**PRIN**

```
PRIN    (<lnum> <lnum>?)?
PRINF   (<lnum> <lnum>?)?
PRINN   (<lnum> <lnum>?)?
```

The PRINt command is exactly the same as the LIST command except that the displayed lines are printed on the printer rather than the CRT. All PRINt variants correspond exactly to the LIST variants. PRINt displays, however, are not paged; <esc> is still in effect.

Example: PRINT

Result: The primary file is printed on the user's printer.

## Program Debugging Commands

ARIAN supports two commands which give the user some assembly language program debugging capabilities. These commands allow the user to set breakpoints and examine the contents of the registers after a breakpoint has been reached. The commands are:

1. BREK -- set, reset, and display the breakpoints.
2. REGS -- set and display register values.

Program Debugging Commands

BREK

```
BREK    <hadr>
BREKD   <hadr>
BREKL
BREKS
```

The breakpoint (BREK) commands permit the user to set, reset, and display the breakpoints. When a breakpoint is reached during a program's execution, the byte replaced by the breakpoint is restored, the address of the breakpoint is displayed to the user, the contents of all the registers are saved in the register save area of ARIAN, and

control is returned to ARIAN. The user may now examine the register contents at his discretion, and, if he wishes to resume program execution, the user may type the CONTinue command.

BREK will set a breakpoint at the specified address; BREKD will clear the breakpoint at the address specified if there is one and inform the user if there is no breakpoint at that address, BREKL will list the addresses of all breakpoints set by the user that have not yet been cleared, and BREKS will clear all user breakpoints that have not yet been cleared.

Example: BREK 4000

Result: A one-byte breakpoint is set at location 4000 hexadecimal. If the user's program later encounters this breakpoint, the message "4000 BREAKPOINT" will be printed.

REGS REGS (<rname> (<val8) ! <val16>))?

The register examine/set (REGS) command allows the user to display the register values in the register save area and reset them if he desires. REGS with no argument will display the values of all the registers. REGS with the arguments will load the specified register or register pair with the specified value. Valid names for the registers are A, F, B, C, D, E, H, L, A', F', B', C', D', E', H', L', IX, IY, and SP. Valid names for the register pairs are BC, DE, HL, BC', DE', and HL'.

Example: REGS BC 10BC

Result: The BC register pair is loaded with the value 10BC. If a CONT is executed, BC will have this value.

## Assembler Commands

The assembler in ARIAN is one of its most useful features, and this pseudo-Z80 assembler is controled through two commands:

1. ASSM -- assemble the primary file.
2. SYMT -- display the assembly or system symbol table.

## Assembler Commands

**ASSM**

```
ASSM    <fname> (<hadr> <hadr>?)?
ASSMF   <fname> (<hadr> <hadr>?)?
ASSML   <fname> (<hadr> <hadr>?)?
ASSMP   <fname> (<hadr> <hadr>?)?
```

The ASSM command instructs ARIAN to assemble the primary file. If <fname> is specified, an entry is made in the binary file directory which specifies the boundaries of the object code generated by the assembler. If no address is given, the object code generated by the assembler is placed immediately after the local text file workspace; if one address is given, the object code is placed starting at this address; if two addresses are given, the object code is assembled to execute at the first address and it is physically placed at the second.

ASSM assembles the primary file and lists only the lines with errors in them. ASSML lists the file in paged mode on the user's CRT as it is being assembled, ASSMF lists the file in formated paged mode on the user's CRT as it is being assembled, and ASSMP prints the file on the user's printer as it is being assembled.

In all cases, the boundaries and length of the object code produced are displayed on the user's CRT.

Example: ASSM T1 5800 6800

Result: The primary file will be assembled to execute at location 5800. The object code produced will be placed at 6800, and the local binary file T1, which defines the limits of this assembly in terms of the 5800 address will be produced.

Example: ASSML

Result: The primary file is assembled at the end of the workspace. All lines with their object code are listed on the user's CRT. For example, if the workspace is defined to be 2000 to 5000 hexadecimal, the object code will be placed starting at 5001.

**SYMT** SYMT
SYMTS

The SYMT command displays the symbol table produced in the last assembly if the system has not been reset. SYMT displays the user program's symbol table, while SYMTS displays the symbol table of the system-defined subroutines.

The following is a list of the system-defined subroutines and a brief description of their use. All I/O routines are routed through the customized I/O driver (normally set for the user's CRT) unless they use the printer. The register usage of these routines is described in Chapter 6 under the corresponding Utility Package routine names.

1. ZEOR -- the reentry point to ARIAN. By entering a JMP ZEOR instruction into his program, the user can accomplish a clean return to ARIAN, preserving the environment of ARIAN at the time of program execution (the local files and customized commands are saved).

2. ZLIN -- this subroutine is the ARIAN system line editor. Calling this subroutine invokes this line editor, putting the entered line into the input buffer.

3. ZINK -- this subroutine polls the <esc> key on the user's CRT and, if <esc> has been typed, executes a clean return to ARIAN.

4. ZIN -- this subroutine inputs one character from the user's CRT keyboard into the A register.

5. ZOUT -- this subroutine outputs one character from the A register to the user's CRT screen.

6. ZCR -- this subroutine outputs a <cr> <lf> to the user's CRT screen.

7. ZARG -- this subroutine is the ARIAN command and argument parser. It may be used in conjunction with ZLIN to input a command and parse it for the user's program. The numeric

arguments entered through ZARG are placed into the binary buffer.

8. ZHOT -- this subroutine displays the A register as two hexadecimal characters on the user's CRT.

9. ZDOT -- this subroutine displays the A register as three decimal digits on the user's CRT.

10. ZBLK -- this subroutine prints a <sp> on the user's CRT.

11. ZPRH -- this subroutine prints the string of ASCII characters pointed to by the H&L register pair on the user's CRT. This string must be terminated by a <cr> (0DH).

12. ZPPH -- this is the same as ZPRH, except that the string is printed on the user's printer.

13. ZPRR -- this subroutine prints the string of ASCII characters pointed to by the return address of the subroutine call. This string must be terminated by a <null> (0).

14. ZPHL -- this subroutine prints the content of the H&L register pair on the user's CRT as four hexadecimal characters.

15. ZBOF -- this symbol points to the two-byte buffer which contains the pointer to the beginning of the current primary file.

16. ZEOF -- this symbol points to the two-byte buffer which contains the pointer to the end of the current primary file.

17. ZBBF -- this symbol points to the binary buffer which contains the values of the binary arguments entered into ZARG.

18. ZIBF -- this symbol points to the 160-byte input line buffer.

19. ZCHA -- this subroutine converts the low nybble (4 bits) of the A register to the ASCII equivalent of this hexadecimal value (0-9, A-F).

20. ZCAH -- this subroutine converts the ASCII hexadecimal character (0-9, A-F) to the binary value it represents.

21. ZEN -- this subroutine exchanges the nybbles of the A register.

22. ZSHD -- this subroutine evaluates B&C = H&L - D&E (the 16-bit difference between the values in the H&L and D&E register pairs; this difference is placed in the B&C register pair). H&L and D&E are unchanged by ZSHD.

23. ZJRL -- this subroutine uses the two bytes pointed to by the return address as the relative displacement from the next instruction to branch to. It is a jump relative long instruction simulator. No register is affected by this subroutine, and the next instruction executed by the user's program is determined by the relative displacement. The DW pseudo-op with an operand field of the form

    LABEL-$-2

    may be used to compute the required displacement in the user's assembly language program.

24. ZCRL -- this subroutine is the same as ZJRL, but it does a call relative long instead of a jump relative long.

Example: SYMT

Result: The symbol table of all the user-defined symbols in the last program assembled with their addresses is displayed on the user's CRT.

## The Utility Command and its Subcommands

The UTIL command gives the user the ability to examine and modify memory directly. The UTIL command itself is of the form:

UTIL

In response to this command, the user is prompted by ARIAN with a slash ("/"). He may then enter the following UTILity subcommands:

1. C <hadr> <hadr> <hadr> -- copy the block of memory defined by the first two addresses to the location of memory starting at the third address. The original block is unchanged unless the third address resides within this block.

2. D <hadr>? <hval>* -- deposit the specified values into memory starting at the given address. If no address is given, the values are deposited starting at the default pointer. When completed, the default pointer points to the next byte to be deposited into.

3. E (<hadr> <hadr>?)? -- examine a specific memory location or a block of memory. The default pointer points to the last location examined. This command is exactly the same as the corresponding command in MCS V1.6.

4. F <hadr> <hadr> <hval>* -- find all occurrances of the specified byte string in the memory block bounded by the addresses given.

5. S <hadr>? -- set/display the default pointer. S with no argument will display the pointer; S with an argument will set the pointer.

6. X -- return to ARIAN command mode.

These commands are a subset of the MCS V1.6 command set, and more details as to their usage may be found in the MCS User's Manual (see [1]).

Example: UTIL

Result: The utility subsystem will be invoked.

Example: E 0 FF

Result: While in the utility subsystem, a block examine is done of locations 0 to FF hexadecimal.

Example: C 0 FF 4000

Result: Block 0 to FF hexadecimal is copied to start at location 4000. As a result, blocks 0 to FF and 4000 to 40FF are identical.

Example: D 2000 0 1 2 3

Result: The values 0, 1, 2, and 3 are deposited into locations 2000 to 2003. The default pointer is now pointing to location 2004.

Example: D 45

Result: Hexadecimal 45 s deposited at location 2004 and the default pointer now points to 2005.

Example: X

Result: UTIL is exited and control is returned to ARIAN.

# CHAPTER 3

## HOW TO USE ARIAN

Like many other operating systems, one learns how to use ARIAN by working and playing with it. This chapter is designed as an introduction to ARIAN, and, with the aid of this chapter, this manual, and ARIAN itself, the user should easily be able to learn how to write and run programs on any micrcomputer system which supports ARIAN.

## How to Execute ARIAN

Execution of ARIAN is a very simple process. After turning on ARIES-I and loading a disk which contains ARIAN as one of its programs, the user should follow the following steps (note: all commands typed are followed by a carriage return):

1. Examine location E900 in the microcomputer's memory. This is the starting address of the FDOS bootstrap PROM.

2. Key the RUN switch on the front panel of ARIES-I. This will run the bootstrap and load FDOS.

3. The phrase "** FDOS V1.3C **" should now be displayed on the CRT screen. If it is not, check to see if the disk is inserted into the drive properly, and if bootstrap still does not work, try using another disk.

4. Once the above message is displayed, the user can execute ARIAN by simply typing "EX ARIAN". If FDOS responds with a "?" and prints its message again, this means that the command was either mistyped or ARIAN does not reside on this disk.

5. If the above command worked, ARIAN should prompt the user with "** ARIAN **", a carriage return, and a command mode prompt (">"). If all went well, ARIAN is now ready to do your bidding.

A second bootstrap system also exists for ARIAN. This second system, the ARIAN Bootstrap Program (ABP), performs a direct bootstrap of ARIAN and also loads FDOS. If the user's disk contains ABP, only steps 1 and 2 from above are performed by the user.

## The ARIAN Input Line Editor

One of the first things the user should know about ARIAN is how to give it a command. This is done by typing the command on the principal CRT keyboard of the microcomputer. Once ARIAN has given the ">" prompt, it is in command mode; it is ready for the user to type a command to it.

All typing done while in ARIAN is processed by the ARIAN Input Line Editor. This editor collects each character as the user types it and allows the user to correct any typing errors he has made. As each character is typed, it is checked to see if it is a special control character. If it is, the function of the control character is executed; if it is not, the character is saved in ARIAN's input line buffer and printed on the user's CRT. When the user has finished typing his line and is satisfied that it is correct, he may then type a carriage return (<cr>), which is a special control character that tells the line editor to finish inputting the line and to give the line to ARIAN to interpret and execute.

The following is a list of all the control characters recognized by the ARIAN Input Line Editor:

1. the Escape (<esc>) key. When typed, <esc> is printed on the CRT as a dollar sign ("$") followed by a <cr>. This key tells the editor to delete the line typed in so far and start over with a new line.
2. the Line Feed (<lf>) key. This key echoes as a <cr> and does not affect the line contained in the input line buffer. The

sole purpose of this function is to allow the user to continue typing his line on the next physical line of the CRT.

3. the Tab (<tab>) or ctrl-I key. This key causes the cursor to tab to the next tab stop. As the cursor is tabbing, spaces are copied into IBUF.

4. the Backspace (<bs>) or ctrl-H key. This key allows the user to delete the last character he typed. It echoes as the cursor backing up to the previous position on the screen. For example, if the user has typed "ABCD<bs>", only "ABC" is in the input line buffer; the "D" has been deleted. If <bs> is typed again, the "C" will be deleted, and so on. The user cannot delete beyond the beginning of his line; if he does, an <esc> is processed, echoing as a "$" <cr>.

5. the Delete (<del>) or Rubout key. This key performs the same thing that <bs> does, but it echoes differently. The deleted characters are enclosed in backslashes. For instance, if the user typed "ABCD<del>E", this would be echoed as "ABCD\D\E", indicating that the D was deleted and the string in the input buffer is now "ABCE". If the user types more than one <del> in a row, all the deleted characters are enclosed in one set of backslashes. For example, if the user types "ABCDE<del><del><del>ABE<del>C", this will appear on his terminal as "ABCDE\EDC\ABE\E\C", indicating that "EDC" and then "E" were deleted and the resulting string is "ABABC". This feature is provided primarily to permit the use of a device that does not support a hardware backspace to be used as the principal I/O device.

6. the Carriage Return (<cr>) key. Again, the <cr> key always instructs the editor to terminate the input of the line and give the line to ARIAN to interpret.

The input line editor is used in every aspect of ARIAN except for the intra-line editing mode (see the EDIT command), and these commands are in effect whenever the user is typing something. This editor is an extremely useful tool, and with practice it will soon become very easy and natural to use.

## How to Create a Local File

One of the first things the user wishes to use ARIAN for is to write an assembly language program and execute it. In order to do this, he must know how to create a file. This is done in a number of ways.

The first and easiest way is to use the FILE command. By typing FILE <fname>, like FILE MYPROG, the user can let ARIAN create a file for him. In response to this command, ARIAN will figure out where to put the file in memory, initialize the file, and respond with something like

MYPROG 2A00 2A00

This indicates that ARIAN has initialized the file and set it to start at location 2A00 in memory. Now, to type his program into this file, the user need only use the APND command. By typing APND, the user instructs ARIAN that he wants to add a block of lines to the end of the current primary file (the file he just created). ARIAN will then respond with a "?" prompt and permit the user to type the lines of his program. All text files in ARIAN must be numbered, but the APND command puts the user in block line entry mode, which automatically numbers the line for the user. At this point, the user simply types the program text, and, when he has finished, types a ctrl-C followed by a <cr>. ARIAN will then renumber the file and place the block of lines just entered into the file.

Example: The following is an example of a short program entered by the user. From now on, underline phrases or symbols in the examples presented indicate that these were typed by ARIAN, and the reset is typed by the user.

```
** ARIAN **
>FILE TEST
TEST 2A00 2A00
? MVI A,1
?LOOP OUT 0FFH ; OUTPUT TO PORT FF
? RLC
? JMP LOOP ; DO IT FOREVER ^C
>LIST
0010  MVI A,1
0020 LOOP OUT 0FFH ; OUTPUT TO PORT FF
0030  RLC
0040  JMP LOOP ; DO IT FOREVER
>
```

Result: The FILE command created the file TEST at location 2A00 and the user then entered lines into this file using the APND command. Note that he terminated the entering of these lines by typing a ctrl-C (^C) followed by a <cr> (carriage returns are not shown in the above example). He then instructed ARIAN to list the file, and it did, showing the line numbers it assigned to the lines of the file. ARIAN automatically inserts a space between the line number and the first character typed in each line.

This example shows the most frequently-used method of creating a file in ARIAN. Other types of files, such as the binary files mentioned in Chapter 2 of this manual, may also be created at the user's

discretion.

## How to Assemble and Execute a Program

Now that the program has been written, the user probably wants to assemble and execute it. This may be done in a number of ways.

In order to quickly assemble and execute a program, the user may simply type EXEC. This command will then assemble and execute the primary file (the program just typed). Another option of the user is to first explicitly assemble his program and then execute it. This may be done by typing ASSM followed by EXECB. ASSM will assemble the program, and EXECB will begin execution at the default binary address (this address was just set by the assembler). More details on the specifics of the assembler are given in Chapter 5.

Example: Now that the user has entered his program, the following steps are taken to run it.

```
>EXEC
ASM PASS 1
ASM PASS 2
6800 6807 0008
```

Result: The EXEC command automatically invoked the assembler. The two ASM PASS messages were printed by the assembler, indicating what it was doing, and the assembler finally told the user that the object code for his program resides from 6800 to 6807 hexadecimal and it is 8 bytes long. His program is now running, and he should be seeing a changing bit pattern on the front panel LEDs.

Example: Another example of the same thing is:

```
>ASSM
ASM PASS 1
ASM PASS 2
6800 6807 0008
>EXECB
```

Result: This does the same thing that the EXEC command did before. The front panel LEDs should now be displaying a changing bit pattern.

## How to Interrupt a Rampaging Program

By now the user should have noticed that his display is running far too fast for him to see what is going on and that he cannot easily regain control of ARIAN. In reality, the latter is not the case.

Whenever the user wishes to gain control of ARIAN after something has gone wrong, there are three ways to do it:

1. stop the computer, examine location 0, and key the RUN switch. This will restart ARIAN, but all the user's local files will be lost and can only be restored by using the RCVR command.

2. stop the computer, examine location 4, and key the RUN switch. This also will restart ARIAN, but this time no program is lost and the environment of ARIAN is almost completely unaffected by the restart. This action performs the RESEt command manually (see the RESE command in Chapter 2).

3. key the EXTERNAL CLEAR switch on the front panel of ARIES-I. This switch has been specifically wired to execute a non-maskable interrupt with ARIAN. Control is returned to ARIAN. A reset is not done, and nothing is changed in ARIAN provided that the program that ran wild did not alter ARIAN.

From time to time, the program under test will destroy all or part of ARIAN. When this happens, the restart procedures described above may not work, and the user may have to bootstrap ARIAN in again.

## How to Save and Load Programs on Disk

If the user feels that his program may not work right the first time, he may wish to save it on disk before he tries to execute it. That way, if the program destroys ARIAN, he can recover easily. Also, the user may wish to save his programs when he has finished with them or he has to go away and shut down ARIES-I for some reason.

Saving, and later loading, programs is done very easily in ARIAN. In order to save a program, the user may simply type the SAVE <fname> command, like SAVE MYPROG. ARIAN will then save the primary file, regardless of what its local name is, on disk under the name specified ("MYPROG"). Later, when the user returns to the system and wishes to reload his program, he may simply type the LOAD <fname> command, like LOAD MYPROG. The specified file is made primary, and he may go on using it as he normally would.

Example: An example of saving and loading programs follows.

```
>SAVE IT
FILE SAVED
>LOAD IT
IT          2A3F 2A3F
>
```

Result: In the above example, the primary file, MYPROG, was saved on disk under the name "IT" and reloaded. Two files now exist in

memory -- IT and MYPROG. Both files are exactly the same, but IT is the primary file. The addresses given above indicate the creation of another file upon the load. If a local file with the name "IT" already existed, ARIAN would prompt the user with "REPLACE?", to which he would respond with "Y" to load over the local file and "N" to abort the load.

## Summary

The above examples show the user how to basically use ARIAN to do what he wants to do, but this is not all ARIAN can do. Chapter 2 gives a complete explanation of all of the commands available to the user and their variants, and the following chapters describe certain commands in more detail. Again, it should be emphasized that the user learns to work with ARIAN by sitting down and doing things with it and generally playing with it. It is a forgiving system, and, as the user will later discover, it will usually allow him to make mistakes and recover from his mistakes.

# CHAPTER 4

## THE CUST COMMAND AND ITS USAGE

The CUST command is a useful command in ARIAN, but it is somewhat complicated to use. This command allows the user to extend and modify the normal set of ARIAN commands, changing the functions of the normal ARIAN commands, and creating his own unique set of commands.

## How to Create a Customized Command

Customized commands are created by the CUST <cname> <hadr>? variant. The subroutine starting at the specified or implied address becomes the new command, and, until the user deletes this customized command, whenever he types the command name given, he will execute this subroutine.

Example: CUST LIST

Result: The LIST command of ARIAN is redefined, and the user will execute the subroutine starting at the default address whenever he types "LIST".

Example: CUSTD LIST

Result: The user-defined LIST command is deleted, and the normal system LIST command is restored.

Example: CUST TEST 6C00

Result: A new customized command called TEST is created; its execution starts at location 6C00 hexadecimal.

Example: CUSTL

Result: All the customized commands defined by the user that are currently in effect are listed and their execution addresses are displayed.

## How to Use Customized Commands Effectively

Effective use of the customized commands the user defines requires the user to understand command parsing in ARIAN. Commands are parsed in ARIAN according to several distinct command subfields (see the SDL description of a command in Chapter 2). All commands start with a command name followed by one or two special characters and any number of arbitrary characters. The command name is always four characters long (the DEL command is actually DEL <sp>), and the special characters are

the fifth and sixth characters typed. The special characters are placed in memory locations IBUF+4 and IBUF+5, resp., by the parser.

The second command subfield (optional) is a file or command name. This field consists of up to eight characters, the first of which must be alphabetic. This field is stored in FBUF, left justified and blank filled. If the argument is a command name, only the first four characters are significant and the rest of the characters are stored but should be ignored by the customized command.

The third command subfield is the numeric argument field. This field consists of from one to three numbers, the first of which must start with a digit if there is no file name given in the command. The arguments parsed in this field are placed in ABUF (four ASCII characters per argument) and BBUF (two bytes per value). ABUF contains the ASCII characters of the numbers in groups of four (i.e., ABUF to ABUF+3 contains the first number, ABUF+4 to ABUF+7 contains the second, and ABUF+8 to ABUF+11 contains the third); BBUF contains their values, assuming they are hexadecimal numbers, in groups of two (i.e., BBUF and BBUF+1 contain the first value in low order-high order, BBUF+2 and BBUF+3 contain the second, and BBUF+4 and BBUF+5 contain the third).

Therefore, with the arguments of a customized command parsed in this manner, the user can create customized commands which perform like the normal ARIAN system commands. ABUF is usually used to contain line numbers for reference, BBUF is usually used to contain values used to

specify a limit, and FBUF is usually used to contain a file or command name.

Example: REGS BC 2000

Result: This is a normal ARIAN command. When it is parsed and interpreted, FBUF contains "BC" for the interpreting subroutine and BBUF contains the value 2000 hexadecimal. The command name "REGS" is in IBUF to IBUF+3.

Example: DDIRP

Result: This ARIAN command is stored in IBUF to IBUF+3, and the special character "P" is stored in IBUF+4 for interpretation.

The user can create his own customized commands to use the argument parser. With practice, it becomes very easy to use the buffers provided by the parser to obtain the desired arguments. This is perhaps the most effective way to use the CUST command.

Example: CUST UPLOAD

Result: This is an example of a user-defined customized command which may use the buffers described. This command is designed to be used with the following format: "UPLO <fname>". Here, the parser will parse out the indicated file name, and, in this example, the user will then locate the specified local file and upload its text to the external computer through a modem.

Example: CUST TEST

Result: Another way the user can use the customized command feature of ARIAN is to define the default assembly and execution address with a command name. This enables him to run the subroutine at this address by simply typing "TEST" rather than "EXECB", or, rather, he may have two or more areas into which he is loading object code, and a different customized command may be created for each area. In the latter case, EXECB is not sufficient to handle more than one area at a time.

# CHAPTER 5

## THE ARIAN ASSEMBLER

The assembler of ARIAN is a very powerful real-time assembler based on the INTEL standard 8080 mnemonics. However, the assembler is not just an 8080 assembler; it is a pseudo-Z80 assembler which gives the user the ability to to assemble some of the common Z80 instructions as well as all of the 8080 instructions.

This assembler features its own set of pseudo-ops (most of which are similar to the INTEL standard pseudo-ops), all the 8080 mnemonics, some arithmetic operations in the operand field, literal character definitions in the operand field, a unique set of Z80 instructions, the ability to explicitly create binary files, and manipulation of the default execution address.

The standard features supported by the assembler include:

1. free format source input
2. symbolic addressing, including forward and relative symbolic references
3. up to 384 six-character symbols
4. reserved names for the registers

## The Assembler in General

The assembler translates the lines contained in the primary file into object code. The second character following the line number is the first source code character position. Therefore, the character immediately following the line number should normally be a space; the APND and INS commands place a space here automatically, and the user need only be concerned with this restriction if he enters his own lines using the <lnum> <text> command. Line numbers are not processed by the assembler; they are merely reproduced in the listing.

The assembler will assemble a source program file composed of statments, comments, and pseudo operations on each line. It does this in two passes. During Pass 1, the assembler allocates all storage necessary for the translated program and defines the values of all

symbols used by creating a symbol table. The storage allocated for the object code will begin at the byte explicitly or implicitly specified by the ASSM command unless an ORG pseudo-op is present in the program. During Pass 2, all expressions, symbols, and ASCII constants are evaluated and placed in allocated memory in the appropriate locations. The listing, also produced during Pass 2, indicates exactly what data is in each location of memory.

Statements contain either symbolic ARIAN assembly machine instructions or pseudo-ops. The structure of such a statement is: (1) name, (2) operation, (3) operand, and (4) comment.

The name field, if present, must begin in the first assembler character position; this is the second character after the line number. The symbol in the name field can contain as many characters as the user desires, but only the first six characters are used in the symbol table to uniquely define the symbol. All symbols in this filed must begin with an alphabetic character and may contain no special characters. Digits are allowed.

The operation field contains either an ARIAN assembler operation mnemonic or a system pseudo-op. The ARIAN assembler operation mnemonics and system pseudo-ops are described below.

The operand field contains parameters pertaining to the operation in the operation field. If two arguments are present, they must be

separated by a comma. All fields are separated and distinguished from one another by one or more spaces.

The comment field is for explanatory remarks. It is reproduced in the listing without processing. Comment lines must start with either a semicolon or an asterisk; it is recommended that comments at the end of a statement also start with one of these characters, but this is not a restriction.

Symbolic names and addressing are also supported by the assembler. To assign a symbolic name to a statement, the name is placed in the name field. To leave off the name field, the user skips two or more spaces after the line number (one or more spaces in block line entry mode) and begins the operation field. If a name is attached to a statement, the assembler assigns it the value of the current location (program) counter. The program counter holds the address of the next byte to be assembled if the instruction is a machine instruction or pseudo-op. The EQU pseudo-op, however, assigns to its label a value which is defined in the operand field. Note: do not confuse the location counter of the assembler with the "$" symbol discussed later; this location counter's value points to the next instruction to be assembled, while "$" points to the instruction after the current instruction if the current instruction is a normal mnemonic or "$" points to the current instruction if it is a pseudo-op.

Names are defined when they appear in the name, or label, field. All defined names may be used as symbolic arguments in the operand field. The reserved system symbols, however, are defined by the assembler and must not be redefined by the user; a duplicate label error will result if this is done. These reserved system symbols are discussed later and in Chapter 2.

In addition to the user-defined and the system symbols, the assembler has reserved several symbols used to represent the registers of the 8080. These symbols, like the system reserved symbols, may only be used in the operand field. These symbols are:

1. A -- the accumulator; value 7.
2. B -- the B register; value 0.
3. C -- the C register; value 1.
4. D -- the D register; value 2.
5. E -- the E register; value 3.
6. H -- the H register; value 4.
7. L -- the L register; value 5.
8. M -- memory (pointed to by H&L); value 6.
9. P -- the program status word; value 6.
10. PSW -- also the program status word.
11. S -- the stack pointer; value 6.
12. SP -- also the stack pointer.

The assembler also supports relative symbolic addressing. If the name of a particular location is known, a nearby location may be specified using the known name and a numeric offset. All defined symbols, including "$", may be used in this relative symbolic addressing mode.

Example: LDA $+5

Result: This instruction loads the accumulator with the value of the byte located five bytes after the beginning of the next instruction.

Example: SSPD LOC-7

Result: This instruction stores the value of the stack pointer starting at the byte located seven bytes in front of the memory location pointed to by the symbol "LOC".

The assembler permits the user to write positive and negative numbers directly in a statement. They will be regarded as integer

constants, and their binary values will be used appropriately. All unsigned numbers are considered to be positive. Decimal constants can be defined using the suffix "D" after the numeric value, but this is not required since the default is decimal. Hence, 10 and 10D define the constant ten decimal. Hexadecimal constants must start with a digit and end with the suffix "H". Examples of hexadecimal constants are 10H, 0AFH, 000101H, and 00BCH.

ASCII constants may be defined by enclosing the ASCII character within single quotes, i.e., 'C'. Two characters may be enclosed within single quotes for double word constants.

## Assembler Pseudo-ops

The following is a list and a description of the pseudo-ops recognized by the assembler:

1. ORG <operand> -- set the origin at the specified address. This instruction also resets the execution address, the assembly limits, and the location in memory at which the object code is loaded. If an ORG appears in the program anywhere but the beginning, the limits set by the last ORG are reflected in the execution address and assembly limits.

2. DS <operand> -- define storage. This reserves the specified number of bytes starting at the current location of the program counter.

3. DB <operand> -- define one byte. This instruction evaluates the specified operand and loads one 8-bit value into the location pointed to by the program counter.

4. DW <operand> -- define one word. This instruction evaluates the specified operand, producing a 16-bit value which it loads into memory (low order, high order) at the location pointed to by the program counter.

5. ASC '<string>' -- ASCII string. This is the same as DB, but the specified string is loaded into memory.

6. <label> EQU <operand> -- the specified label is assigned the computed value of the operand. The computed value is a 16-bit quantity.

7. END -- end the assembly. This statement is not absolutely required; assembly will stop when the end of the file is reached.

All pseudo-ops may be preceeded by a label.

## System Reserved Labels

Another feature of the ARIAN assembler is its system reserved labels (these labels are described in Chapter 2 under the SYMT command). These labels provide a host of utility subroutines for functions such as I/O, data conversion, and ARIAN entry points, and they also supply some commonly-used buffer and variable addresses. All system reserved labels start with the letter "Z" and are at most four characters long.

## The Special Z80 Mnemonics

The following is a list of the special Z80 mnemonics recognized by the ARIAN assembler and their ZILOG equivalents.

1. SSPD <operand> -- ZILOG mnemonic: LD (nn),SP (store SP direct)
2. LSPD <operand> -- ZILOG mnemonic: LD SP,(nn) (load SP direct)
3. SBCD <operand> -- ZILOG mnemonic: LD (nn),BC (store BC direct)
4. LBCD <operand> -- ZILOG mnemonic: LD BC,(nn) (load BC direct)
5. SDED <operand> -- ZILOG mnemonic: LD (nn),DE (store DE direct)
6. LDED <operand> -- ZILOG mnemonic: LD DE,(nn) (load DE direct)
7. EXA -- ZILOG mnemonic: EXA
8. EXX -- ZILOG mnemonic: EXX
9. BR <operand> -- ZILOG mnemonic: JR n (branch relative)
10. BC <operand> -- ZILOG mnemonic: JR C,n (branch relative on carry)
11. BZ <operand> -- ZILOG mnemonic: JR Z,n (branch relative on zero)
12. BNC <operand> -- ZILOG mnemonic: JR NC,n (branch relative on no carry)
13. BNZ <operand> -- ZILOG mnemonic: JR NZ,n (branch relative on no zero)
14. DBJ <operand> -- ZILOG mnemonic: DJNZ n (decrement b and jump relative)
15. LD -- ZILOG mnemonic: LDD
16. LDR -- ZILOG mnemonic: LDDR
17. LI -- ZILOG mnemonic: LDI
18. LIR -- ZILOG mnemonic: LDIR
19. CD -- ZILOG mnemonic: CPD

20. CDR -- ZILOG mnemonic: CPDR

21. CI -- ZILOG mnemonic: CPI

22. CIR -- ZILOG mnemonic: CPIR

23. NEG -- ZILOG mnemonic: NEG

24. RLD -- ZILOG mnemonic: RLD

25. RRD -- ZILOG mnemonic: RRD

26. SHB -- ZILOG mnemonic: SBC HL,BC

27. SHD -- ZILOG mnemonic: SBC HL,DE

28. SHS -- ZILOG mnemonic: SBC HL,SP

29. AHB -- ZILOG mnemonic: ADC HL,BC

30. AHD -- ZILOG mnemonic: ADC HL,DE

31. AHS -- ZILOG mnemonic: ADC HL,SP

32. IM0 -- ZILOG mnemonic: IM 0

33. IM1 -- ZILOG mnemonic: IM 1

34. IM2 -- ZILOG mnemonic: IM 2

35. ID -- ZILOG mnemonic: IND

36. IDR -- ZILOG mnemonic: INDR

37. II -- ZILOG mnemonic: INI

38. IIR -- ZILOG mnemonic: INIR

39. OD -- ZILOG mnemonic: OUTD

40. ODR -- ZILOG mnemonic: OTDR

41. OI -- ZILOG mnemonic: OUTI

42. OIR -- ZILOG mnemonic: OTIR

43. CIN -- ZILOG mnemonic: IN A,(C)

44. COT -- ZILOG mnemonic: OUT (C),A

## Operand Evaluation

Operand evaluation is somewhat limited in ARIAN, particularly due to the size restrictions on the system. Parenthesized expressions are not permitted. Only sixteen-bit addition and subtraction are permitted in infix expressions. Single character strings of the form '<char>' are permitted in expressions and unarily.

All numeric arguments are assumed to be decimal unless the suffix "H" is appended to them. Therefore, 100 is decimal 100 and 100H is hexadecimal 100.

The "$" symbol is used as the value of the program counter. In normal instructions, "$" points to the first byte of the next instruction; in pseudo-ops, "$" points to the first byte of the pseudo-op. This permits relative addressing to take the form of "BR LABEL-$" and pseudo-ops like "STACK EQU $" to be used.

Finally, if an expression with a value greater than 0FF hexadecimal is loaded into an eight-bit register, like "MVI A,1FFH", only the low-order byte of this value is loaded.

Examples of permitted expressions include:

```
LABEL+3
POINT-'A'+60
POINT3-0AFH+6-2
HERE-$-2
```

## Assembler Error Messages

The following is a list of the error messages produced by the assembler and their meanings:

1. R -- register error. The register name is missing or invalid.
2. S -- syntax error. The instruction syntax is incorrect.
3. U -- undefined symbol. The referenced symbol is undefined.
4. V -- value error. The computed value cannot be represented as a 16-bit integer or the expression has a syntax error.
5. M -- missing label error. A required label is missing.
6. A -- argument error. The instructions argument is of the wrong type or generally incorrect.
7. L -- label error. The label of this instruction contains an invalid character.
8. D -- duplicate label error. The label of this instruction has been defined elsewhere.
9. O -- opcode error. The opcode in this instruction is invalid.

# CHAPTER 6

## THE ARIES-I UTILITY PACKAGE

The ARIES-I Utility Package is a PROM resident series of support subroutines designed to provide the common functions of I/O, code conversion, memory manipulation, and line editing. This package, residing in two INTEL 2708 PROMs, occupies 2K bytes of memory and contans sixty-three utility subroutines.

These subroutines are grouped into the following catagories:

1. the Link program
2. I/O set/reset programs
3. I/O routines
4. specialized print routines
5. string print routines
6. register print routines
7. query I/O routines

8. move utilities

9. conversion and computation utilities

10. tabulation and related routines

11. an input line editor

12. a command and argument parser (MCS level)

13. call and jump relative utilities

14. an additional print routine

15. tape I/O routines

16. timing loop

These utilities in detail are:

1. the link program

    1. LINK -- address D000; this is exactly the same as the TERM command described in Chapter 2. This subroutine executes that command.

2. I/O set/reset utilities

    1. INIRP -- address D003; this subroutine resets the primary I/O port.

    2. INIP -- address D006; this subroutine sets the primary I/O port for polling use and resets the customized I/O driver.

    3. INIR2 -- address D009; this subroutine resets the secondary I/O port.

    4. INI2 -- address D00C; this sets the secondary for polling use.

    5. SETCI -- address D00F; this subroutine sets the customized input driver. The customized input flag is set and the user must provide the address of the input subroutine in the H&L register pair.

    6. RESETCI -- address D012; this resets the customized input flag, restoring the normal input routine.

7. SETCO -- address D015; this sets the customized output driver. The customized output flag is set and the user must provide the address of the output subroutine in the H&L register pair.

8. RESETCO -- address D018; this resets the customized output flag, restoring the normal output routine.

3. I/O routines

1. INPUT -- address D01B; this is the primary input routine. The customized and primary input routines are controlled by this program. The A register contains the value input through this routine.

2. OUTPUT -- address D01E; this is the primary output routine. The customized and primary output routines are controlled by this program. The user must place the value to output in the A register.

3. INPB -- address D021; this is the same as INPUT, but the input value is contained in both the A and B registers.

4. OUTB -- address D024; this is the same as OUTPUT, but the value to be output must be contained in the B register.

5. INP2 -- address D027; this routine inputs from the secondary input port into the A register.

6. OUT2 -- address D02A; this routine outputs the value in the A register to the secondary output port.

7. INP2B -- address D02D; this is the secondary port equivalent of INPB.

8. OUT2B -- address D030; this is the secondary port equivalent of OUTB.

9. INP3 -- address D033; this is the tertiary port equivalent of INP2.

10. OUT3 -- address D036; this is the tertiary port equivalent of OUT2.

11. INP3B -- address D039; this is the tertiary port equivalent of INPB.

12. OUT3B -- address D03C; this is the tertiary port equivalent of OUTB.

4. the specialized print routines

1. CRLF -- address D03F; this routine prints a <cr> <lf> through OUTPUT.

2. PRBL -- address D042; this routine prints a space (blank) through OUTPUT.

3. PCC -- address D045; this routine prints "^C" through OUTPUT.

4. BSLSH -- address D048; this routine prints a backslash through OUTPUT.

5. the string print routines

   1. PRINT -- address D04B; this prints the string starting at the return address and ending in a <null>. This subroutine returns to the byte following the <null>.

   2. PPRINT -- address D04E; this is the same as PRINT, but the string is printed on the printer.

   3. SCRN -- address D051; this prints the string pointed to by H&L through OUTPUT; the string must end in a <cr> character.

   4. SCRNC -- address D054; this is the same as SCRN, but a carriage return is printed after the string is printed.

   5. PEM -- address D057; this prints the two ASCII characters immediately following its call through OUTPUT; a "**" is printed in front of these characters, and it is designed to be used to print two-character error messages. This subroutine returns to the byte following the second character.

   6. PCHAR2 -- address D05A; this is the same as PEM, but the "**" is not printed.

   7. PCHAR -- address D05D; this is the same as PCHAR2, but only one character is printed.

6. the register print routines

   1. PA2HC -- address D060; the value contained in the A register is printed as two hexadecimal ASCII characters throught OUTPUT.

   2. PA3DC -- address D063; the value contained in the A register is printed as three decimal ASCII characters through OUTPUT.

3. PA2DC -- address D066; the value contained in the A register is printed as two decimal ASCII characters through OUTPUT. This value must be less than 100 decimal.

4. PADC -- address D069; the value contained in the A register is printed as the minimum required number of decimal ASCII characters to represent it.

5. PHL -- address D06C; the value contained in the H&L register pair is printed as four hexadecimal ASCII characters through OUTPUT.

6. PHLB -- address D06F; this is the same as PHL, but a blank is printed after the characters.

7. PHL5D -- address D072; the value contained in the H&L register pair is printed as five decimal ASCII characters through OUTPUT.

8. PDE -- address D075; this prints the value contained in the D&E register pair as four hexadecimal ASCII characters through OUTPUT.

7. the query I/O routines

1. PQ -- address D078; this prints " Q?" through OUTPUT, inputs one character from the user through INPUT, and compares it with "N" and returns.

2. INK -- address D07B; this checks the primary I/O port to see if a character was typed, and, if it was, it inputs the character into the A register, compares it with <esc>, and returns.

8. the move utilities

1. LMOV -- address D07E; this moves characters from the location addressed by D&E to the location addressed by H&L until the value in the C register is encountered. This character is not moved. These pointers are incremented until the specified value is encountered, and, when finished, D&E point to this value and H&L point to the position the last character is moved to.

2. LMOVC -- address D081; the character string pointed to by D&E is moved to the location pointed to by H&L. The number of characters moved is contained in the C register, and the pointers are incremented while this move occurs.

3. LMOVL -- address D084; this is the same as LMOVC, except the number of characters to be moved is defined in B&C.

4. RMOV -- address D087; this is the same as LMOV, except the pointers are decremented instead of incremented.

5. RMOVC -- address D08A; this is the same as LMOVC, except the pointers are decremented.

6. RMOVL -- address D08D; this is the same as LMOVL, except the pointers are incremented.

7. MOVE -- address D090; this moves the block pointed to by D&E to start at the location pointed to by H&L. The block is B&C bytes long. This subroutine decides whether to use LMOVL or RMOVL.

9. the conversion and computation utilities

1. CHTA -- address D093; this routine converts the low nybble of the A register into its equivalent ASCII representation, also stored in the A register.

2. CATH -- address D096; this routine converts the ASCII hexadecimal digit into its binary equivalent low nybble of the A register. If A contains an invalid character upon entry, it will contain the value 20 hexadecimal upon exit as an error indicator.

3. EN -- address D099; the high and low nybbles of the A register are interchanged.

4. HLMDE -- address D09C; the expression B&C = H&L - D&E is evaluated. H&L and D&E are unchanged.

5. RANGE -- address D09F; this computes B&C = D&E - H&L +1. H&L and D&E are unchanged.

10. tabulation and related routines

1. TABS -- address D0A2; this routine uses SCALE and READ to allow the user to set the system tab stops. See Chapter 2 for a description of this under the TABS command.

2. TABE -- address D0A5; this uses SCALE and OUTPUT to allow the user to examine (displays) the system tab stops as they are currently set.

3. TABR -- address D0A8; this sets the default system tab stops (every 8 columns).

4. SCALE -- address D0AB; this prints the column scale across the CRT screen.

11. the input line editor

1. READ -- address D0AE; this is the system input line editor used by ARIAN.

12. a command and argument parser

    1. GETIN -- address D0B1; this routine is a simple one-character command and argument parser similar to that used by MCS. The first non-blank character is passed out in the A register, and up to two 16-bit hexadecimal values are converted to binary and passed out in the D&E and H&L register pairs. H&L contains the first value encountered, and D&E contains the second. If there is only one or no values given, the corresponding values passed are zero. This routine inputs its command string through the input line editor.

    2. GETV -- address D0B4; this routine searches for the first non-blank character starting at the location pointed to by H&L and converts the valid hexadecimal ASCII characters which follow into a 16-bit value in the D&E register pair. Computation is stopped on the first invalid character, and H&L point to this character when done.

13. the call and jump relative utilities

    1. CREL -- address D0B7; this routine has been described in Chapter 2 under the SYMT command. It is pointed to by the ZCRL symbol.

    2. JREL -- address D0BA; this routine has been described in Chapter 2 under the SYMT command. It is pointed to by the ZJRL symbol.

14. an additional print routine

    1. PRNTL -- address D0BD; this routine prints the string pointed to by H&L and terminated by a <cr>; it will return to the byte following the string

15. tape I/O routines

    1. TSTRT13 -- address D0C0; this routine starts cassette tape drive 1 and sets the baud rate for this drive at 300 baud; a CUTS board is required to support this software

    2. TSTRT112 -- address D0C3; same as TSTRT13, but the speed is 1200 baud

    3. TSTRT23 -- address D0C6; same as TSTRT13, but the drive number is 2

4. TSTRT212 -- address D0C9; same as TSTRT112, but the drive number is 2

5. TSTOP3 -- address D0CC; this routine stops all drives and sets their speeds at 300 baud

6. TSTOP12 -- address D0CF; this routines stops all drives and sets their speeds at 1200 baud

7. TWRITE -- address D0D2; this routine writes the byte contained in the A register on the drive currently running

8. TREAD -- address D0D5; this routine reads the byte coming in from the drive currently running and places it into the A register

16. timing loop

1. DELAY2S -- address D0D8; this routine is a delay loop which counts for approximately two seconds; it may be used to pause after tape start-up to allow the read and write heads to settle and the tape to get to speed

# APPENDIX II -- OBJECT CODE FORMATS

# CHAPTER 1

## OBJECT CODE OUTPUT DEFINITIONS

ARIES-I uses several object code formats when it communicates with a host processor for cross-assembly downloading and other microcomputers for program transfer. These formats provide a convenient method for data communication over mediums such as paper tape and telephone in which a seven-bit ASCII code must be used to transmit eight-bit information. This chapter describes these formats in detail and discusses them briefly.

The object code formats recognized by the LINK program and ARIES-I are the ARIAN paper tape format (derived from the MUMS format [2]) and the INTEL standard object code format [3].

## 1.1 The ARIAN Paper Tape Format

The ARIAN paper tape format is the simplest of the object code formats described in this chapter. It permits downloading of a block of data with the minimum amount of overhead information; only the load address, the data, and a checksum are presented by this object code format.

The information is presented to the downloading system in the form of a group of load blocks separated by '@' characters (hexadecimal 40). A load block is defined as one unit of data which is to be loaded into sequential memory locations and a group of overhead information which defines the load block and its disposition. Such overhead information commonly includes the address at which the data load is to start and a checksum which permits some verification that the transmitted data is correct.

The SDL definition of the ARIAN paper tape format is as follows:

```
('@'* '#' <ahd> <ahd> <ahd> <ahd>

(<ahd> <ahd> <cr> <lf>)* '&' <ahd> <ahd> '$'?  '@'*)+
```

As the reader can see, the ARIAN paper tape format is divided into the following frames:

1. opening header -- This is the string of '@' characters at the beginning of the load block.

2. load block designator -- The '#' character at the beginning of the load block tells the downloader that a load block follows.

3. load address -- The four ASCII hexadecimal digits immediately following the load block designator represent the address at which the loading of the data in the block is to begin. This address is specified as any N16 hexadecimal value would be written; i.e., 'EABC' would indicate hexadecimal address EABC.

4. data -- The data is represented by pairs of ASCII hexadecimal digits in the same way the load address is represented. Carriage returns and line feeds may be interspersed among the data pairs to improve legibility of the load block and permit limits imposed by the host computer (such as limited line length) to be met without forcing restrictions on the size of the load block.

5. checksum indicator and value -- A checksum is included at the end of each load block to enable downloading system to verify the accuracy of the data received. This checksum is preceeded by a '&' character. The checksum is the ASCII representation of the modulo-256 sum of the binary values represented by each pair of ASCII hexadecimal digits transmitted. This sum is computed by adding the high-order execution address value, the low-order execution address value, and the data values in modulo-256.

6. termination character -- A '$' should be placed at the end of the group of load blocks to be downloaded to indicate that no more load blocks follow.

7. closing leader -- A string of '@' characters may be placed at the end of a load block for convenience in using paper tape.

A loader for this object code format should execute in the following sequence:

1. First, it should loop until it receives a load block designator ('#').

2. Secondly, it should zero the local checksum byte and decode the next four ASCII characters as an address, updating the checksum value as it decodes.

3. The loader should then decode each pair of characters received, add this value to the checksum, and deposit this value into the appropriate memory location and increment the address pointer until an '&' character is received.

4. Finally, it should decode the two checksum digits received, subtract them from the local checksum byte, and begin looking for the next load block designator ('#') or termination character ('$'). If a checksum error occurs (the difference between the transmitted checksum and the local checksum byte is not zero), the loader should give the user an error message and stop the download.

## 1.2 The INTEL Standard Object Code Format

The INTEL Standard Object Code Format, also known as the MOSTEK Z80 Object Output Definition of Type 0, is described in SDL as follows:

(':' (<ahd> <ahd>)*3 '00' (<ahd> <ahd>)+ <cr> <lf>)+

As can be seen by the above description, this data format is divided into the following fields:

1. delimiter -- this is the colon which begins each record

2. number of bytes in current record -- the number of bytes in the current record is given immediately after the colon. Represented by two hexadecimal characters, the maximum value allowed is 32 decimal (20 hexadecimal).

3. load address -- the load address is given by the following four ASCII hexadecimal digits (high order, low order).

4. data type -- the data type is 0, represented by two ASCII 0's.

5. data -- the data is represented by the following pairs of ASCII hexadecimal digits. The number of pairs is given by the byte count in the second field.

6. checksum -- the last two ASCII hexadecimal digits represent the checksum. This checksum is the negative of the binary sum of all bytes in the record.

7. <cr> <lf> -- all records are terminated by a <cr> <lf> pair.

# CHAPTER 2

## EXAMPLES of the OBJECT CODE FORMATS

The following strings are examples of the ARIAN Object Code Format:

1. '#E0200001&01' -- load at E020 hexadecimal a binary 0 and at E021 a binary 1.

2. '#10001010&30' -- load at 1000 and 1001 hexadecimal a value of hexadecimal 10.

The following strings are examples of the INTEL Standard Object Code Definition:

1. ':021000000001ED' -- load at 1000 and 1001 hexadecimal 0 and 1, resp.

2. ':0000000000' -- load nothing at location 0.

# APPENDIX III -- STRING DESCRIPTION LANGUAGE (SDL)

String Description Language, hereafter referred to as SDL, is a language definition which enables one to describe the format and syntax of character strings. SDL was developed by the author as a result of his lack of satisfaction with other string description languages such as BNF and Extended BNF [4]; SDL is a combination of what the author considers to be the good points of several string description languages and the features he finds lacking in them. This appendix is a description of SDL as used in this paper.

## SDL Variables and Constants

All constants in SDL are string constants, which take the form of zero or more characters of the ASCII character set enclosed in single quotes. The double single quote ('') is used to represent a single quote constant. Examples of SDL constants are 'abc', 'hello', ' ', '0', '1234abc', and '*:bc1,.'.

A variable in SDL is a symbol which may be assigned a value. Two types of variables are used in SDL -- terminal and non-terminal variables. A non-terminal variable is a string of alphanumeric characters enclosed in angle brackets (<>), and it is used to represent certain special characters (such as carriage return) and characters or strings the user wishes to define in his SDL equations. A terminal variable is a string of alphanumeric characters (the first of which must be alphabetic) which is not enclosed in angle brackets. Terminal variables are used to represent user-defined strings. The difference between non-terminal and terminal variables is that non-terminal variables may not necessarily be defined in the user's string definition while terminal variables must always be defined in the definition. Non-terminal variables which are not defined in the user's string definition are assumed to be mnemonic in nature or SDL-defined symbols. The following is a list of the non-terminal variables which are SDL-defined symbols.

1. <cr> or <CR> -- the ASCII carriage return character
2. <lf> or <LF> -- the ASCII line feed character
3. <colon> or <COLON> -- the ASCII colon character (:)
4. <star> or <STAR> -- the ASCII asterisk character (*)

5. <bang> or <BANG> -- the ASCII exclamation mark (!)

6. <quote> or <QUOTE> -- the ASCII single quote (')

7. <dquote> or <DQUOTE> -- the ASCII double quote (")

8. <plus> or <PLUS> -- the ASCII plus symbol (+)

9. <minus> or <MINUS> -- the ASCII minus symbol (-)

10. <lparen> or <LPAREN> -- the ASCII left parentheses symbol

11. <rparen> or <RPAREN> -- the ASCII right parentheses symbol

12. <ques> or <QUES> -- the ASCII question mark (?)

13. <adig> or <ADIG> -- an ASCII digit character ('0' ... '9')

14. <ahd> or <AHD> -- an ASCII hexadecimal digit character ('0' ... '9', 'A' ... 'F')

15. <letter> or <LETTER> -- an ASCII upper-case letter ('A' ... 'Z')

16. <alpha> or <ALPHA> -- an ASCII letter, upper or lower case

17. <digit> or <DIGIT> -- same as <adig>

18. <alphit> or <ALPHIT> -- an <alpha> or a <digit>

19. <blank> or <BLANK> -- an ASCII space (hexadecimal value 20)

20. <null> or <NULL> -- the null or empty string

21. <ellipsis> or <ELLIPSIS> -- the mark of ellipsis (...).

These SDL-defined non-terminal variables may be used in any SDL expression without being defined elsewhere in the body of expressions. These symbols may also be redefined by the user at his discretion, but care must be taken not to think of their original SDL definitions if the redefinition of the variable assigns it a meaning different from its SDL meaning. For example, <blank> may be redefined to be the string '  ' rather than ' ', but the user must be sure that <blank> has the '  '

meaning throughout the SDL text.

As mentioned above, non-terminal variables may also be mnemonic in nature. If the user considers them to be so, he need not define them explicitly in the SDL description he is writing. In so doing, he leaves their definition to the reader. Examples of non-terminals used in this way are:

1. <nameofperson> -- this obviously means the name of a person. This string, however, may be interpreted in many ways; examples are 'Sam Spade', 'S. Spade', 'SPADE', etc.

2. <number> -- a number, which may be interpreted in a very large number of ways, such as '1', '1.333e+4', 'ABCH', '12 * 10', etc.

## SDL Operators

SDL allows the user to employ several operators in the SDL expressions he creates. These operators include the double-quote ("), the asterisk (*), the plus symbol (+), the exclamation mark (!), the question mark (?), the ellipsis (...), and the left and right parentheses. As the user will note, all of these symbols have SDL-defined non-terminal variables which represent them.

The uses of the SDL operators are defined as follows:

1. double-quote -- this is used to enclose a description phrase. A description phrase is a phrase used to describe a variable; an example of a description phrase is "THE DIGITS 0 AND 1".

2. exclamation mark -- this is the logical OR symbol in SDL. Used in expressions, it means that the right-hand side or the left-hand side are possible. For example, ' <alpha> ! <digit> ' says that whatever is being discussed may be an <alpha> or a <digit>.

3. question mark -- this means that the preceeding symbol or expression is optional. For example, ' <alpha>? ' means that an <alpha> is optional.

4. parentheses -- these are used as grouping symbols and may be nested as deeply as desired.

5. asterisk -- this is one of SDL's two multiplication symbols. It means that the preceeding symbol or expression may be repeated zero or more times. For example, ' <digit>* ' means that <digit> may be repeated zero or more times. Hence, strings like '0', '123321', '43147832929387499 8987', and <null> are permitted. If the asterisk is followed by a number, the number specifies the upper limit of the multiplication. For example, ' <digit>*4 ' says that there may be zero to 4 digits in the string.

6. plus -- this is the other multiplication symbol. It is used exactly like asterisk, but it specifies one or more. The number following it may also be present, indicating an upper limit.

7. colon -- this is the assignment symbol. It must be present in every SDL statement, and it assigns the expression on its right to the variable on its left.

8. ellipsis -- this is the mark of ellipsis (...). It is used to imply symbols or characters found in between two specified symbols or characters. For example, ' 'A' ... 'F' ' refers to ' 'A', 'B', 'C', 'D', 'E', 'F' '.

## SDL Statements

An SDL description of a string or set of strings consists of a group of SDL statements. An SDL statement consists simply of a variable, which may be used to represent the string in question, followed by a colon and an SDL expression. An SDL expression is a grouping of variables, operators, description phrases, and ellipses which describes a string. In this grouping each variable is separated by one or more delimiters, where a delimiter is an operator, an angle bracket, or a <blank>.

The following is a group of examples of SDL statements:

1. <hexdigit> : <digit> ! 'A' ... 'F'

   This statement defines <hexdigit> to be a <digit> (defined by SDL) or one of the characters 'A' to 'F'.

2. HEXADDRESS : <hexdigit>*4

HEXADDRESS is defined as a string of 0 to 4 <hexdigit>s. For example, HEXADDRESS may be '', 'ABC', '0123', '00', '0A1B'.

3. NUMBER : <digit>+4

NUMBER is defined as a string of 1 to 4 <digit>s.

4. ITIS : <alpha>+ '.'?

ITIS is defined as a string of one or more <alpha>s followed optionally by a period.

5. ENGLISHSENTENCE : "ANY COMBINATION OF ASCII CHARACTERS"

ENGLISHSENTENCE is defined by a description phrase. Note the generality of the phrase.

6. THING : NUMBER ! (NUMBER <blank>+)+4 NUMBER ! HEXADDRESS*3

THING is defined as (1) one NUMBER, (2) from 1 to 4 NUMBERs, each followed by 1 or more <blank>s, followed by a NUMBER, or (3) from 0 to 3 HEXADDRESSs appended to each other. For example, THING may be '1', '1 234 234', 'ABCDE123332', or '1 1 1 1 1'.

7. NEAT : 'A' ! 'B' ! 'A' NEAT 'B'

NEAT is recursively defined. Strings like 'A', 'B', 'AAB', 'AAAABBB', and 'AABBB' are possible.

The user will note that <blank>s are never implied in SDL. They must be placed in the string explicitly. In the example with "THING," when THING was defined as HEXADDRESS*3, it may have actually been desired to express it as ' (HEXADDRESS <blank>+)*3 ' to separate each HEXADDRESS.

## Summary

SDL is a recursive language which is used to define strings of ASCII characters. It permits implicit or explicit definition of strings, and it can be generally used to describe almost any language or set of strings. Derived from BNF and Extended BNF, it contains many features from these languages as well as several extensions made by the author.

Some features of SDL are:

1. SDL-defined non-terminal variables which need not be defined by the user.
2. mnemonic definition of non-terminal variables
3. the use of ellipsis to indicate items left out of a definition

4. multiplication of symbols with * or + may be followed by a number to indicate the limit of multiplication

5. recursive definition which allows a vriable to be defined in terms of itself

6. the use of a description phrase in a variable definition

# APPENDIX IV -- A SAMPLE ARIAN SESSION

```
>*
>*   THIS IS AN EXAMPLE OF A SESSION WITH ARIAN
>*      REDIRECTABLE I/O WAS USED TO CREATE THIS LISTING
>*
>
>
>*
>*   THE FOLLOWING IS A LISTING OF THE PROGRAM WHICH
>*      REDIRECTED THE I/O FOR THIS LISTING
>*
>FILE
SYSLOG   2A00  2C84
>LISTF
0010              ORG     0F8C0H
0020     UPLOAD   PUSH    PSW  ; SAVE DATA
0030     UPL2     IN      MSTAT
0040              ANI     TBMT
0050              BZ      UPL2-$
0060              POP     PSW
0070              OUT     MDATA
0080              CALL    OUTP
0090              PUSH    PSW
0100              CPI     0DH
0110              BNZ     DONE-$
0120              MVI     A,7FH
0130              CALL    OUTT
0140              CALL    OUTT
0150              CALL    OUTT  Q? Y
0160     DONE     POP     PSW
0170              RET
0180     OUTT     PUSH    PSW
0190     OUTTL    IN      MSTAT
0200              ANI     TBMT
0210              BZ      OUTTL-$
0220              POP     PSW
0230              OUT     MDATA
0240              RET
0250     OUTP     PUSH    PSW
0260     OUTPL    IN      TSTAT
0270              ANI     TBMT
0280              BZ      OUTPL-$
0290              POP     PSW
0300              OUT     TDATA  Q? Y
0310              RET
0320     MSTAT    EQU     6
0330     MDATA    EQU     7
0340     TSTAT    EQU     4
0350     TDATA    EQU     5
```

```
0360      TBMT    EQU     2
0370      COUNT   DS      1
>LIST 10
0010  ORG 0F8C0H
>DEL 10
>WORK
2A00 87FF
>ASSM
ASM PASS 1
ASM PASS 2
8800 8835 0036
>ASSMF
ASM PASS 1
ASM PASS 2
  8800 F5       0020 UPLOAD  PUSH    PSW  ; SAVE DATA
  8801 DB06     0030 UPL2    IN      MSTAT
  8803 E602     0040         ANI     TBMT
  8805 28FA     0050         BZ      UPL2-$
  8807 F1       0060         POP     PSW
  8808 D307     0070         OUT     MDATA
  880A CD2A88   0080         CALL    OUTP
  880D F5       0090         PUSH    PSW
  880E FE0D     0100         CPI     0DH
  8810 200B     0110         BNZ     DONE-$
  8812 3E7F     0120         MVI     A,7FH
  8814 CD1F88   0130         CALL    OUTT
  8817 CD1F88   0140         CALL    OUTT
  881A CD1F88   0150         CALL    OUTT
  881D F1       0160 DONE    POP     PSW  Q?

  881E C9       0170         RET
  881F F5       0180 OUTT    PUSH    PSW
  8820 DB06     0190 OUTTL   IN      MSTAT
  8822 E602     0200         ANI     TBMT
  8824 28FA     0210         BZ      OUTTL-$
  8826 F1       0220         POP     PSW
  8827 D307     0230         OUT     MDATA
  8829 C9       0240         RET
  882A F5       0250 OUTP    PUSH    PSW
  882B DB04     0260 OUTPL   IN      TSTAT
  882D E602     0270         ANI     TBMT
  882F 28FA     0280         BZ      OUTPL-$
  8831 F1       0290         POP     PSW  Q?
  8832 D305     0300         OUT     TDATA
  8834 C9       0310         RET
  0006          0320 MSTAT   EQU     6
  0007          0330 MDATA   EQU     7
  0004          0340 TSTAT   EQU     4
  0005          0350 TDATA   EQU     5
  0002          0360 TBMT    EQU     2
  8835          0370 COUNT   DS      1
8800 8835 0036
```

```
>LDIR
SYSLOG   2A00  2C72
>LDIRB
8800 8835
>*
>*  NOW FOR A DEMO OF THE SOFTWARE DEVELOPMENT CAPABILITIES
>*       OF ARIAN
>*
>LDIR
SYSLOG   2A00  2C72
>FILE DEMO1
DEMO1    2C73  2C73
>APND
?*
?*       THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
?*       LANGUAGE PROGRAMS
?*
?START  CALL     ZPRR     ; PRINT A STRING
?       ASC      'THIS IS A TEST'
?       DB       0
?       RET
?c
>LIST
0010 *
0020 *       THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
0030 *       LANGUAGE PROGRAMS
0040 *
0050 START  CALL     ZPRR     ; PRINT A STRING
0060        ASC      'THIS IS A TEST'
0070        DB       0
0080        RET
>RNUM 100
>LIST
0100 *
0110 *       THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
0120 *       LANGUAGE PROGRAMS
0130 *
0140 START  CALL     ZPRR     ; PRINT A STRING
0150        ASC      'THIS IS A TEST'
0160        DB       0
0170        RET
>RNUM 100 100
>LIST
0100 *
0200 *       THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
0300 *       LANGUAGE PROGRAMS
0400 *
0500 START  CALL     ZPRR     ; PRINT A STRING
0600        ASC      'THIS IS A TEST'
0700        DB       0
0800        RET
>EXEC
```

```
ASM PASS 1
ASM PASS 2
8800 8812 0013
THIS IS A TEST
>LDIR
DEMO1    2C73  2D5B
SYSLOG   2A00  2C72
>LDIRB
8800 8812
>*
>*  NOW FOR ANOTHER PROGRAM
>*
>FILE DEMO2
DEMO2    2D5C  2D5C
>LDIR
DEMO2    2D5C  2D5C
DEMO1    2C73  2D5B
SYSLOG   2A00  2C72
>APND
?*
?*       ANOTHER DEMO
?*
? MVI A,30 ; I'LL PRINT THE CHARACTERS '0' - '9'
? MVI B,10
?LOOP CALL ZOUT ; PRINT CHAR IN A
? INR A ; PREPARE NEXT CHAR
? DBJ LOOP-$
? RETc
>LIST
0010 *
0020 *       ANOTHER DEMO
0030 *
0040  MVI A,30 ; I'LL PRINT THE CHARACTERS '0' - '9'
0050  MVI B,10
0060 LOOP CALL ZOUT ; PRINT CHAR IN A
0070  INR A ; PREPARE NEXT CHAR
0080  DBJ LOOP-$
0090  RET
>LISTF
0010      *
0020      *                    ANOTHER DEMO
0030      *
0040              MVI     A,30 ; I'LL PRINT THE CHARACTERS '0' - '9'
0050              MVI     B,10
0060      LOOP    CALL    ZOUT ; PRINT CHAR IN A
0070              INR     A ; PREPARE NEXT CHAR
0080              DBJ     LOOP-$
0090              RET
```

```
 !"#$%&'
>*  OOOPS!  SHOULD HAVE BEEN 30 HEXADECIMAL!
>LISTF
0010      *
0020      *                    ANOTHER DEMO
0030      *
0040              MVI     A,30 ; I'LL PRINT THE CHARACTERS '0' - '9'
0050              MVI     B,10
0060      LOOP    CALL    ZOUT ; PRINT CHAR IN A
0070              INR     A ; PREPARE NEXT CHAR
0080              DBJ     LOOP-$
0090              RET
>40  MVI A,30H ; I'LL PRINT '0'-'9'
>LISTF
0010      *
0020      *                    ANOTHER DEMO
0030      *
0040              MVI     A,30H ; I'LL PRINT '0'-'9'
0050              MVI     B,10
0060      LOOP    CALL    ZOUT ; PRINT CHAR IN A
0070              INR     A ; PREPARE NEXT CHAR
0080              DBJ     LOOP-$
0090              RET
>EXEC
ASM PASS 1
ASM PASS 2
8800 880A 000B
0123456789
>*  NOW, LET'S PUT A <SP> BETWEEN EACH DIGIT
>INS 80
? PUSH PSW
? CALL ZBLK
? POP PSWc
>LISTF
0010      *
0020      *                    ANOTHER DEMO
0030      *
0040              MVI     A,30H ; I'LL PRINT '0'-'9'
0050              MVI     B,10
0060      LOOP    CALL    ZOUT ; PRINT CHAR IN A
0070              INR     A ; PREPARE NEXT CHAR
0080              PUSH    PSW
0090              CALL    ZBLK
0100              POP     PSW
0110              DBJ     LOOP-$
0120              RET
>EXEC
ASM PASS 1
ASM PASS 2
8800 880F 0010
0 1 2 3 4 5 6 7 8 9
>*  I LIKE IT!  I LIKE IT!
```

```
>LDIR
DEMO2    2D5C  2E51
DEMO1    2C73  2D5B
SYSLOG   2A00  2C72
>*  NOW, BACK TO DEMO1
>FILE DEMO1
DEMO1    2D69  2E51
>LDIR
DEMO1    2D69  2E51
DEMO2    2C73  2D68
SYSLOG   2A00  2C72
>*  NOTE THE MEMORY MANAGER
>LISTF
0100     *
0200     *                  THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
0300     *                  LANGUAGE PROGRAMS
0400     *
0500     START          CALL    ZPRR    ; PRINT A STRING
>LIST
0100 *
0200 *      THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
0300 *      LANGUAGE PROGRAMS
0400 *
0500 START  CALL    ZPRR    ; PRINT A STRING
0600        ASC     'THIS IS A TEST'
0700        DB      0
0800        RET
>610  DB 0DH
>620  ASC 'AND THIS IS ONLY A TEST'
>LIST
0100 *
0200 *      THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
0300 *      LANGUAGE PROGRAMS
0400 *
0500 START  CALL    ZPRR    ; PRINT A STRING
0600        ASC     'THIS IS A TEST'
0610  DB 0DH
0620  ASC 'AND THIS IS ONLY A TEST'
0700        DB      0
0800        RET
>*  WHOOPS!
>DEL 610 620
>LIST 600 700
0600        ASC     'THIS IS A TEST'
0700        DB      0
>INS 700
?       DB      0DH
?       ASC     'AND THIS IS ONLY A TEST'c
>LISTF 600 9999
>RNUM 100 100
>LIST 600 9999
0600        ASC     'THIS IS A TEST'
```

```
0700          DB      0DH
080C          ASC     'AND THIS IS ONLY A TEST'
0900          DB      0
1000          RET
>*  NOTE THAT THE INS COMMAND RENUMBERED THE FILE!
>EXEC
ASM PASS 1
ASM PASS 2
8800 882A 002B
THIS IS A TEST
AND THIS IS ONLY A TEST
>*  I LIKE IT!  I LIKE IT!
>LDIR
DEMO1     2D69  2E99
DEMO2     2C73  2D68
SYSLOG    2A00  2C72
>FILE DEMO3
DEMO3     2E9A  2E9A
>APND
?*
?*  ANOTHER DEMO
?*
?LOOP CALL ZIN  ; GET CHAR FROM PIODEV
? PUSH PSW
? CALL ZPRR
? ASC 'YOUR CHARACTER WAS: '
? DB 0
? POP PSW
? CALL ZOUT
? PUSH PSW
? CALL ZCR  ; <CR> <LF>
? CALL ZPRR
? ASC 'ITS ASCII VALUE IS: '
? DB 0
? POP PSW
? PUSH PSW
? CALL ZHOT
? CALL ZPRR
? ASC 'IN HEX AND '
? DB 0
? POP PSW
? CALL ZDOT
? CALL ZPRR
? ASC 'IN DECIMAL.'
? DB 0
? CALL ZCR
? BR LOOP-$
?c
>LISTF
0010      *
0020      *               ANOTHER DEMO
0030      *
```

```
0040     LOOP    CALL    ZIN  ; GET CHAR FROM PIODEV
0050             PUSH    PSW
0060             CALL    ZPRR
0070             ASC     'YOUR CHARACTER WAS: '
0080             DB      0
0090             POP     PSW
0100             CALL    ZOUT
0110             PUSH    PSW
>APND 40
? CPI 3  ; ABORT IF CTRL-C
? RZ
>LISTF
0010     *
0020     *               ANOTHER DEMO
0030     *
0040     LOOP    CALL    ZIN  ; GET CHAR FROM PIODEV
0050             CPI     3  ; ABORT IF CTRL-C
0060             RZ
0070             PUSH    PSW
0080             CALL    ZPRR
0090             ASC     'YOUR CHARACTER WAS: '
0100             DB      0
0110             POP     PSW
0120             CALL    ZOUT
0130             PUSH    PSW
0140             CALL    ZCR  ; <CR> <LF>
0150             CALL    ZPRR    Q? Y
0160             ASC     'ITS ASCII VALUE IS: '
0170             DB      0
0180             POP     PSW
0190             PUSH    PSW
0200             CALL    ZHOT
0210             CALL    ZPRR
0220             ASC     'IN HEX AND '
0230             DB      0
0240             POP     PSW
0250             CALL    ZDOT
0260             CALL    ZPRR
0270             ASC     'IN DECIMAL.'
0280             DB      0
0290             CALL    ZCR
0300             BR      LOOP-$
>*  AS AN EXAMPLE, WHICH LINES PUSH OR POP THE PSW?
>FIND
SEARCH STRING? PSW
0070  PUSH PSW  Q? Y
0110  POP PSW
0130  PUSH PSW
0180  POP PSW
0190  PUSH PSW
0240  POP PSW
>EXEC
```

```
ASM PASS 1
ASM PASS 2
8800 886A 006B
YOUR CHARACTER WAS: T
ITS ASCII VALUE IS: 54IN HEX AND 084IN DECIMAL.

>*  WHOOPS!  NO LIKE!
>FIND
SEARCH STRING? IN HEX
0220  ASC 'IN HEX AND '
>220  ASC ' IN HEX AND '
>FIND
SEARCH STRING? IN DEC
0270  ASC 'IN DECIMAL.'
>270  ASC ' IN DECIMAL.'
>ASSMF
ASM PASS 1
ASM PASS 2
  8800           0010 *
  8800           0020 *  ANOTHER DEMO
  8800           0030 *
  8800 CD1BD0    0040 LOOP    CALL    ZIN  ; GET CHAR FROM PIODEV
  8803 FE03      0050         CPI     3  ; ABORT IF CTRL-C
  8805 C8        0060         RZ
  8806 F5        0070         PUSH    PSW
  8807 CD4BD0    0080         CALL    ZPRR
  880A           0090         ASC     'YOUR CHARACTER WAS: '
  881E 00        0100         DB      0
  881F F1        0110         POP     PSW
  8820 CD1ED0    0120         CALL    ZOUT
  8823 F5        0130         PUSH    PSW
  8824 CD3FD0    0140         CALL    ZCR  ; <CR> <LF>  Q? Y
  8827 CD4BD0    0150         CALL    ZPRR
  882A           0160         ASC     'ITS ASCII VALUE IS: '
  883E 00        0170         DB      0
  883F F1        0180         POP     PSW
  8840 F5        0190         PUSH    PSW
  8841 CD60D0    0200         CALL    ZHOT
  8844 CD4BD0    0210         CALL    ZPRR
  8847           0220         ASC     ' IN HEX AND '
  8853 00        0230         DB      0
  8854 F1        0240         POP     PSW
  8855 CD63D0    0250         CALL    ZDOT
  8858 CD4BD0    0260         CALL    ZPRR
  885B           0270         ASC     ' IN DECIMAL.'
  8867 00        0280         DB      0
  8868 CD3FD0    0290         CALL    ZCR  Q? Y
  886B 1893      0300         BR      LOOP-$
8800 886C 006D
>LDIR
DEMO3    2E9A  30D2
DEMO1    2D69  2E99
```

```
DEMO2    2C73  2D68
SYSLOG   2A00  2C72
>LDIRB
8800 886C
>EXECB
YOUR CHARACTER WAS: T
ITS ASCII VALUE IS: 54 IN HEX AND 084 IN DECIMAL.
YOUR CHARACTER WAS: L
ITS ASCII VALUE IS: 4C IN HEX AND 076 IN DECIMAL.
YOUR CHARACTER WAS: H
ITS ASCII VALUE IS: 48 IN HEX AND 072 IN DECIMAL.
YOUR CHARACTER WAS: E
ITS ASCII VALUE IS: 45 IN HEX AND 069 IN DECIMAL.
YOUR CHARACTER WAS: L
ITS ASCII VALUE IS: 4C IN HEX AND 076 IN DECIMAL.
YOUR CHARACTER WAS: L
ITS ASCII VALUE IS: 4C IN HEX AND 076 IN DECIMAL.
YOUR CHARACTER WAS: O
ITS ASCII VALUE IS: 4F IN HEX AND 079 IN DECIMAL.

>*  HOW ABOUT A PROMPT?
>LIST
0010 *
0020 *  ANOTHER DEMO
0030 *
0040 LOOP CALL ZIN  ; GET CHAR FROM PIODEV
0050  CPI 3  ; ABORT IF CTRL-C
0060  RZ
0070  PUSH PSW
0080  CALL ZPRR
0090  ASC 'YOUR CHARACTER WAS: '
>EDIT 40
 0040 LOOP CALL ZIN  ; GET CHAR FROM PIODEV
?0040 \LOOP\ CALL ZIN  ; GET CHAR FROM PIODEV
>LIST 30 50
0030 *
0040  CALL ZIN  ; GET CHAR FROM PIODEV
0050  CPI 3  ; ABORT IF CTRL-C
>INS 40
?LOOP CALL ZPRR
? ASC 'CHARACTER? '
? DB 0
?c
>LIST
0010 *
0020 *  ANOTHER DEMO
0030 *
0040 LOOP CALL ZPRR
0050  ASC 'CHARACTER? '
0060  DB 0
0070  CALL ZIN  ; GET CHAR FROM PIODEV
0080  CPI 3  ; ABORT IF CTRL-C
```

```
0090  RZ
0100  PUSH PSW
0110  CALL ZPRR
0120  ASC 'YOUR CHARACTER WAS: '
0130  DB 0
0140  POP PSW
0150  CALL ZOUT  Q? N
>INS 110
? CALL ZCRc
>ASSMF
ASM PASS 1
ASM PASS 2
  8800           0010 *
  8800           0020 *  ANOTHER DEMO
  8800           0030 *
  8800 CD4BD0    0040 LOOP    CALL    ZPRR
  8803           0050         ASC     'CHARACTER? '
  880E 00        0060         DB      0
  880F CD1BD0    0070         CALL    ZIN  ; GET CHAR FROM PIODEV
  8812 FE03      0080         CPI     3  ; ABORT IF CTRL-C
  8814 C8        0090         RZ
  8815 F5        0100         PUSH    PSW
  8816 CD3FD0    0110         CALL    ZCR
  8819 CD4BD0    0120         CALL    ZPRR
  881C           0130         ASC     'YOUR CHARACTER WAS: '
  8830 00        0140         DB      0  Q?
  8831 F1        0150         POP     PSW
  8832 CD1ED0    0160         CALL    ZOUT
  8835 F5        0170         PUSH    PSW
  8836 CD3FD0    0180         CALL    ZCR  ; <CR> <LF>
  8839 CD4BD0    0190         CALL    ZPRR
  883C           0200         ASC     'ITS ASCII VALUE IS: '
  8850 00        0210         DB      0
  8851 F1        0220         POP     PSW
  8852 F5        0230         PUSH    PSW
  8853 CD60D0    0240         CALL    ZHOT
  8856 CD4BD0    0250         CALL    ZPRR
  8859           0260         ASC     ' IN HEX AND '
  8865 00        0270         DB      0
  8866 F1        0280         POP     PSW
  8867 CD63D0    0290         CALL    ZDOT  Q? Y
  886A CD4BD0    0300         CALL    ZPRR
  886D           0310         ASC     ' IN DECIMAL.'
  8879 00        0320         DB      0
  887A CD3FD0    0330         CALL    ZCR
  887D 1881      0340         BR      LOOP-$
8800 887E 007F
>LDIRB
8800 887E
>*  AS AN EXAMPLE, I WILL NOW CREATE A CUSTOMIZED COMMAND
>CUSTL
>*  NOTE -- NO CUSTOMIZED COMMANDS CURRENTLY EXIST
```

```
>CUST PLAY
>CUSTL
PLAY 8800
>PLAYCHARACTER?
YOUR CHARACTER WAS: T
ITS ASCII VALUE IS: 54 IN HEX AND 084 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: E
ITS ASCII VALUE IS: 45 IN HEX AND 069 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: N
ITS ASCII VALUE IS: 4E IN HEX AND 078 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: X
ITS ASCII VALUE IS: 58 IN HEX AND 088 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: 3
ITS ASCII VALUE IS: 33 IN HEX AND 051 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: 9
ITS ASCII VALUE IS: 39 IN HEX AND 057 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: [
ITS ASCII VALUE IS: 5B IN HEX AND 091 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: ;
ITS ASCII VALUE IS: 3B IN HEX AND 059 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: _
ITS ASCII VALUE IS: 5F IN HEX AND 095 IN DECIMAL.
CHARACTER?
>LDIR
DEMO3    2E9A  3118
DEMO1    2D69  2E99
DEMO2    2C73  2D68
SYSLOG   2A00  2C72
>LDIRB
8800 887E
>CUSTL
PLAY 8800
>*  NOW, EVERYTIME I TYPE 'PLAY', THE PROGRAM AT 8800H WILL
>       BE EXECUTED
>LDIR
DEMO3    2E9A  3118
DEMO1    2D69  2E99
DEMO2    2C73  2D68
SYSLOG   2A00  2C72
>FILE DEMO2
DEMO2    3023  3118
>LDIR
DEMO2    3023  3118
DEMO1    2C73  2DA3
```

```
DEMO3     2DA4  3022
SYSLOG    2A00  2C72
>ASSM
ASM PASS 1
ASM PASS 2
8800 880F 0010
>PLAY0 1 2 3 4 5 6 7 8 9
>PLAY0 1 2 3 4 5 6 7 8 9
>FILE DEMO1
DEMO1     2FE8  3118
>ASSM
ASM PASS 1
ASM PASS 2
8800 882A 002B
>PLAYTHIS IS A TEST
AND THIS IS ONLY A TEST
>FCHK
$ VALID FILE
>LDIR
DEMO1     2FE8  3118
DEMO2     2EF2  2FE7
DEMO3     2C73  2EF1
SYSLOG    2A00  2C72
>FCHK DEMO2
$ VALID FILE
>FCHK DEMO3
$ VALID FILE
>FCHK SYSLOG
$ VALID FILE
>LDIR
DEMO1     2FE8  3118
DEMO2     2EF2  2FE7
DEMO3     2C73  2EF1
SYSLOG    2A00  2C72
>UTIL
/E 2A00 2BCF
2A00 23 30 30 32  30 20 55 50  4C 4F 41 44  20 50 55 53
2A10 48 20 50 53  57 20 20 3B  20 53 41 56  45 20 44 41
2A20 54 41 0D 14  30 30 33 30  20 55 50 4C  32 20 49 4E
2A30 20 4D 53 54  41 54 0D 10  30 30 34 30  20 20 41 4E
2A40 49 20 54 42  4D 54 0D 11  30 30 35 30  20 20 42 5A
2A50 20 55 50 4C  32 2D 24 0D  0F 30 30 36  30 20 20 50
2A60 4F 50 20 50  53 57 0D 11  30 30 37 30  20 20 4F 55
2A70 54 20 4D 44  41 54 41 0D  11 30 30 38  30 20 20 43
2A80 41 4C 4C 20  4F 55 54 50  0D 10 30 30  39 30 20 20
2A90 50 55 53 48  20 50 53 57  0D 0F 30 31  30 30 20 20
2AA0 43 50 49 20  30 44 48 0D  12 30 31 31  30 20 20 42
2AB0 4E 5A 20 44  4F 4E 45 2D  24 0D 11 30  31 32 30 20
2AC0 20 4D 56 49  20 41 2C 37  46 48 0D 11  30 31 33 30
2AD0 20 20 43 41  4C 4C 20 4F  55 54 54 0D  11 30 31 34
2AE0 30 20 20 43  41 4C 4C 20  4F 55 54 54  0D 11 30 31   Q? Y
2AF0 35 30 20 20  43 41 4C 4C  20 4F 55 54  54 0D 13 30
```

```
2B00 31 36 30 20  44 4F 4E 45  20 50 4F 50  20 50 53 57
2B10 0D 0B 30 31  37 30 20 20  52 45 54 0D  14 30 31 38
2B20 30 20 4F 55  54 54 20 50  55 53 48 20  50 53 57 0D
2B30 15 30 31 39  30 20 4F 55  54 54 4C 20  49 4E 20 4D
2B40 53 54 41 54  0D 10 30 32  30 30 20 20  41 4E 49 20
2B50 54 42 4D 54  0D 12 30 32  31 30 20 20  42 5A 20 4F
2B60 55 54 54 4C  2D 24 0D 0F  30 32 32 30  20 20 50 4F
2B70 50 20 50 53  57 0D 11 30  32 33 30 20  20 4F 55 54
2B80 20 4D 44 41  54 41 0D 0B  30 32 34 30  20 20 52 45
2B90 54 0D 14 30  32 35 30 20  4F 55 54 50  20 50 55 53
2BA0 48 20 50 53  57 0D 15 30  32 36 30 20  4F 55 54 50
2BB0 4C 20 49 4E  20 54 53 54  41 54 0D 10  30 32 37 30
2BC0 20 20 41 4E  49 20 54 42  4D 54 0D 12  30 32 38 30
/$
* NOTE THE DUMP OF THE FILE "SYSLOG" AS IT EXISTS IN MEMORY**IN
/F 2A00 2C72 33 35
2C40
/E 2C40 2C41
2C40 33 35
/; NOTE THE RESULT OF THE FIND COMMAND
/E 7000 700F
7000 47 54 55 4E  54 0D 13 35  36 35 35 20  57 54 41 50
/E 8000 800F
8000 7E 45 15 E0  8B B3 22 51  0A 8D E0 0F  E2 8F 3B 43
/D 8000 0 0 0 0 0
/D 0 0 0 0 0
/D0 0 0 0 0 0
/E 8000 800F
8000 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
/E8010 801F
8010 F0 AF 95 9D  5A FF F2 F5  AF DF 3D D1  5D BF 68 5B
/; NOTE DEPOSIT AND, FOLLOWING, COPY
/C 8000 800F 8010
/E8000 801F
8000 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
8010 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
/D8008 3 2
/C8000 800F 8010
/E8000 801F
8000 00 00 00 00  00 00 00 00  03 02 00 00  00 00 00 00
8010 00 00 00 00  00 00 00 00  03 02 00 00  00 00 00 00
/; JUST SOME UTILITY EXAMPLES
/X
>LDIR
DEMO1    2FE8  3118
DEMO2    2EF2  2FE7
DEMO3    2C73  2EF1
SYSLOG   2A00  2C72
>LIST
0100 *
0200 *      THIS IS A DEMO OF THE ABILITY TO WRITE ASSEMBLY
0300 *      LANGUAGE PROGRAMS
```

```
>$

>FILE DEMO3
DEMO3    2E9A  3118
>ASSM 8000
ASM PASS 1
ASM PASS 2
8000 807E 007F
>CUST DEMO 8000
>CUSTL
PLAY 8800
DEMO 8000
>DEMOCHARACTER?
YOUR CHARACTER WAS: F
ITS ASCII VALUE IS: 46 IN HEX AND 070 IN DECIMAL.
CHARACTER?
YOUR CHARACTER WAS: X
ITS ASCII VALUE IS: 58 IN HEX AND 088 IN DECIMAL.
CHARACTER?
>LDIR
DEMO3    2E9A  3118
DEMO2    2C73  2D68
DEMO1    2D69  2E99
SYSLOG   2A00  2C72
>FILE DEMO2
DEMO2    3023  3118
>ASSM 8000
ASM PASS 1
ASM PASS 2
8000 800F 0010
>DEMO0 1 2 3 4 5 6 7 8 9
>ASSMF 8000
ASM PASS 1
ASM PASS 2
  8000           0010 *
  8000           0020 *       ANOTHER DEMO
  8000           0030 *
  8000 3E30      0040          MVI     A,30H ; I'LL PRINT '0'-'9'
  8002 060A      0050          MVI     B,10
  8004 CD1ED0    0060 LOOP     CALL    ZOUT ; PRINT CHAR IN A
  8007 3C        0070          INR     A ; PREPARE NEXT CHAR
  8008 F5        0080          PUSH    PSW
  8009 CD42D0    0090          CALL    ZBLK
  800C F1        0100          POP     PSW
  800D 10F5      0110          DBJ     LOOP-$
  800F C9        0120          RET
8000 800F 0010
>INS 40
? CALL ZCR
?c
>LISTF 30 60
0030     *
```

```
0040                CALL    ZCR
0050                MVI     A,30H ; I'LL PRINT '0'-'9'
0060                MVI     B,10
>LISTN 30 60
*
 CALL ZCR
 MVI A,30H ; I'LL PRINT '0'-'9'
 MVI B,10
>ASSM 8000
ASM PASS 1
ASM PASS 2
8000 8012 0013
>DEMO
0 1 2 3 4 5 6 7 8 9
>DEMO
0 1 2 3 4 5 6 7 8 9
>DEMO
0 1 2 3 4 5 6 7 8 9
>*  THIS IS THE END OF THE SAMPLE ARIAN SESSION
>TERM
```